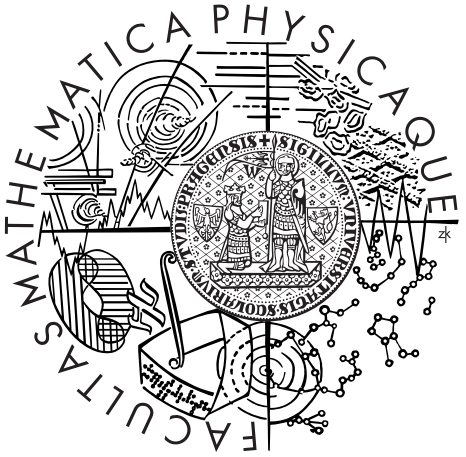
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Jan Klíma

Urychlení evolučních algoritmů pomocí rozhodovacích stromů a jejich zobecnění

Accelerating evolutionary algorithms by decision trees and their generalizations

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: RNDr. Ing. Martin Holeňa CSc.

Study programme: Computer science

Specialization: Theoretical computer science

Prague 2011

Many thanks are addressed to my supervisor, RNDr. Ing. Martin Holeňa, CSc. for his invaluable guidance during the process of writing this thesis. I would also like to thank my parents and all my friends for the support they have given me in the past year.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature

Název práce: Urychlení evolučních algoritmů pomocí rozhodovacích stromů a jejich zobecnění
Autor: Jan Klíma
Katedra: Katedra Teoretické Informatiky a Matematické Logiky
Vedoucí diplomové práce: RNDr. Ing. Martin Holeňa CSc., Ústav Informatiky Akademie Věd České Republiky

Abstrakt: Evoluční algoritmy jsou jednou z nejúspěšnějších metod pro řešení netradičních optimalizačních problémů. Protože evoluční algoritmy používají pouze funkční hodnoty cílové funkce, blíží s k jejímu optimu mnohem pomaleji než optimalizační metody pro hladké funkce. Tato vlastnost evolučních algoritmů je zvláště nevýhodná v kontextu nákladného a časově náročného empirického způsobu získávání hodnot cílové funkce. Evoluční algoritmy však lze podstatně urychlit použitím dostatečně přesného regresního modelu cílové funkce. Cílem práce je výzkum využitelnosti regresních stromů a regresních lesů jako náhradního modelu k urychlení evoluční optimalizace empirických cílových funkcí.

Klíčová slova: evoluční optimalizace, regresní stromy, regresní lesy, náhradní modelování

Title: Accelerating evolutionary algorithms by decision trees and their generalizations
Author: Jan Klíma
Department: Department of Theoretical Computer Science and Mathematical Logic
Supervisor: RNDr. Ing. Martin Holeňa CSc., Institute of Computer Science, Academy of Sciences of the Czech Republic

Abstract: Evolutionary algorithms are one of the most successful methods for solving non-traditional optimization problems. As they employ only function values of the objective function, evolutionary algorithms converge much more slowly than optimization methods for smooth functions. This property of evolutionary algorithms is particularly disadvantageous in the context of costly and time-consuming empirical way of obtaining values of the objective function. However, evolutionary algorithms can be substantially speeded up by employing a sufficiently accurate regression model of the empirical objective function. This thesis provides a survey of utilizability of regression trees and their ensembles as a surrogate model to accelerate convergence of evolutionary optimization.

Keywords: evolutionary optimization, regression trees, tree ensembles, surrogate modelling

Contents

1. Introduction

In practice, people are often challenged by the necessity of solving tasks characterized by the need to find, among all possible solutions, the solution which is optimal (or at least sufficiently close to optimum) for the given problem. These types of problems are entitled *optimization problems*.

Our goal in this thesis will be to solve problems of *mixed optimization*, ie. find global extremes (maximum or minimum) of an objective real-valued function of continuous and categorical variables. We will assume that the objective function is not given by any explicit analytical formula and we are only able to sample it in a finite number of points. Such functions are usually called *black-box* functions or, in the context of obtaining function values by observation or experiment, *empirical* functions.

Evolutionary optimization algorithms employ only function values of the objective function and are often successfully used to solve the forementioned optimization problems. Unfortunately, evolutionary algorithms converge to global optima rather slowly compared to optimization methods for smooth functions and require a large number of samples (hundreds or even thousands) from the domain of the objective function to be evaluated in the process.

Evaluation of the objective function in one particular point may be expensive in terms of time, money or other resources and if this is the case, making decisions about when and in what points to sample the objective function becomes the most critical part of the optimization algorithm.

One way to tackle optimization of expensive empirical functions is through the use of a *surrogate model*. Instead of sampling the original objective function, approximate regression model of the objective function may be constructed; this model can then be used in place of the empirical objective function to increase convergence speed of the evolutionary algorithm.

We may think about the number of evaluations of the empirical objective function as of a *budget*, which we need to invest as wisely as possible to search the domain of the objective function effectively. The better we manage this limited budget, the faster will the evolutionary algorithm converge. Such way of thinking immediately leads to many questions, for instance what are sound strategies of utilizing the surrogate model in the evolutionary optimization or exactly which types of surrogate models are suitable for the purpose and which are not.

1.1 Thesis Contribution

In the past fifteen years, methods for acceleration of evolutionary optimization through surrogate modelling have been proposed for many specific areas of science and engineering. To contribute to the topic of surrogate modelling in evolutionary optimization, the presented thesis focuses on three goals:

- **Proposing a surrogate model for objective functions of both categorical and continuous variables.** To the best knowledge of the author, all available publications focus on models applicable only to objective functions of continuous variables. A simple, but widely used regression model

for functions of both categorical and continuous variables is based on decision trees. Regression trees are piecewise-constant and can be incorporated into more complex ensemble models. Existing methods for regression based on decision trees were reviewed, implemented and tested on both artificial benchmark functions and real-world data.

- **Evolutionary optimization using the proposed surrogate model.** To test the suitability of regression trees and ensemble methods as surrogate models, genetic algorithm is proposed as a part of the thesis. Prototype implementation of the algorithm was tested on benchmark functions and the results are presented at the end of the thesis.

- **Solving a practical problem.** The proposed genetic algorithm is designed to solve constrained problems of mixed optimization. The definition of the optimization task is presented to the algorithm in the form of a syntactical structure obtained by parsing a formal language commonly used in catalysis, given to the author by his thesis supervisor. Correspondingly, regression models based on trees and their ensembles were also tested on data from a hydrocyanic acid synthesis experiment.

1.2 Thesis Structure

Chapter 2 is dedicated to describe wide range of methods employed in this thesis. Selected regression methods based on decision trees and their ensembles created by bagging and boosting are summarized in Section 2.1. Section 2.2 focuses on evolutionary optimization, beginning with basic principles of evolutionary optimization and a brief introduction to genetic algorithms. Existing methods and approaches of utilizing surrogate modelling in evolutionary computation are recapitulated next in Section 2.3.

Based on the nature of constrained optimization problems encountered in catalysis (reviewed in Section 2.4), genetic algorithm is proposed in Chapter 3 that utilizes surrogate model based on regression trees and tree ensembles.

The prototype implementation of the proposed methods is described in Chapter 4, including implementation details (which may also help potential users to swiftly understand the prototype implementation and use it to their own needs) and description of the problems encountered in the implementation process.

Chapter 5 presents results which were obtained when testing the implemented methods. The first part of the chapter is dedicated to benchmarking of tree-based regression models on artificial benchmark functions and real-world data sets. Results of testing of the implemented genetic algorithm follow next.

Chapter 6 closes the thesis, summarizing observed performance of the proposed methods as well as potential for future development.

2. Employed Methods

In this chapter, methods used throughout the thesis will be explained. First, regression methods based on decision trees are presented in Section 2.1. Section 2.2 is devoted to evolutionary optimization algorithms and more particularly, genetic algorithms. Next part of this chapter (Section 2.3) summarizes the role of surrogate modelling in evolutionary optimization and basic strategies of utilizing the surrogate model. Lastly, Section 2.4 shortly describes the character of optimization problems encountered in catalysis.

2.1 Regression Trees and Their Ensembles

In this section, we will present robust methods for regression based on trees, focusing mainly on regression based on tree ensembles. Our goal will be to train a model $\hat{f}$ to accurately fit a real-valued function f of one or more variables, given a finite set of N training samples $\mathcal{D}_T = \{(\mathbf{x}_i, y_i) | \mathbf{x}_i \in dom(f), y_i \in \mathbb{R}\}$.

To actually measure the quality of the obtained fit, we will use squared loss, defined as $Loss(y_i, \hat{f}(\mathbf{x}_i)) = (y_i - \hat{f}(\mathbf{x}_i))^2$. Then we can calculate error of the model on any arbitrary set of N samples $\mathcal{D} = \{(\mathbf{x}_i, y_i) | \mathbf{x}_i \in dom(f), y_i \in \mathbb{R}\}$ as the sum of squares error $\sum_{i=1}^{N} Loss(y_i, \hat{f}(\mathbf{x}_i))$ or as the mean squared error $\frac{1}{N} \sum_{i=1}^{N} Loss(y_i, \hat{f}(\mathbf{x}_i))$.

2.1.1 Regression Trees

Classification and Regression Trees (CART) have been thoroughly studied by Breiman et al. [2]. Regression tree for a function of one or more variables is defined as a binary tree that meets the following requirements:

- Each inner node of the tree represents a decision rule on a single variable. For some variable X, such rule has the form of $X \in subset(dom(X))$ for X categorical or $X < c, c \in \mathbb{R}$ for X numeric.
- A real-valued response $c \in \mathbb{R}$ is assigned to each leaf.

An example of an regression tree is depicted in Figure 2.1.

Given a particular $\mathbf{x} \in dom(f)$, the response value of a regression tree is obtained by following the path given by decision rules, starting at the root. The finite set of all paths from the root to leaves partitions the domain of f into a finite set of M regions R_m. As a consequence, the response value of a regression tree for $\mathbf{x} \in dom(f)$ may be written as:

$$\hat{f}(\mathbf{x}) = \sum_{m=1}^{M} c_m I(\mathbf{x} \in R_m),$$

where

$$I(\mathbf{x} \in A) = \begin{cases} 1 & \text{if } \mathbf{x} \in A \\ 0 & \text{if } \mathbf{x} \notin A. \end{cases}$$

The formal definition of a regression tree allows us to extend the notation. In the following text, $T(\mathbf{x})$ will always denote the response value of a regression tree

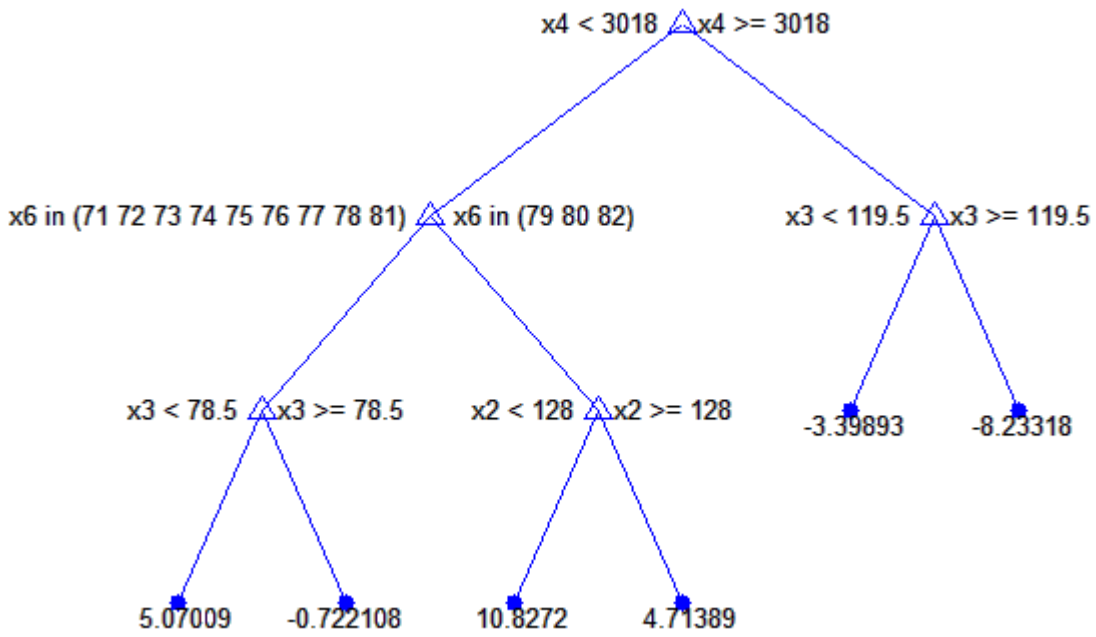


Figure 2.1: An example of a regression tree showing decision rules on three variables, taken from *MATLAB*. x_2 and x_3 are continuous variables, while x_6 is a categorical variable.

and it will be interchangeable with $\hat{f}(\mathbf{x})$ in these situations where $\hat{f}$ also represents a regression tree. T alone will stand for the set of vertices of a regression tree. For node $t \in T$, let us denote by $\mathcal{D}(t)$ the subset of the data set $\mathcal{D}$ which fits into node $t \in T$ and, if t is leaf, $c(t)$ the response value assigned to that leaf. $\tilde{T}$ will denote the set of all leaves within tree T.

For a given data set $\mathcal{D}$ of N samples, sum of squares error of a tree T is defined as $SSE(T) = \sum_{n=1}^{N}(y_i - T(\mathbf{x}_i))^2$. Mean squared error of a tree T is defined as $MSE(T) = \frac{1}{N}\sum_{n=1}^{N}(y_i - T(\mathbf{x}_i))^2$. We can apply the same formulas locally so that for a node $t \in T$, $SSE(t)$ and $MSE(t)$ are errors computed for the subtree given by t on $\mathcal{D}(t)$.

Construction

Since the problem of constructing regression tree with the minimal prediction error is in general computationally infeasible, the construction proceeds in a greedy manner. In each step, children are created for one leaf node l (we call this process *splitting*, because data samples from $\mathcal{D}_T(l)$ are split between its two children) and the best decision rule for node l is chosen from the set of all rules based only on values of $\mathbf{x}$ present in $\mathcal{D}_T(l)$. The whole algorithm:

1. Start with root node r for which $\mathcal{D}_T(r)$ equals $\mathcal{D}_T$ and set $c(r) = mean(y_i|(\mathbf{x}_i, y_i) \in \mathcal{D}_T(r))$.
2. Repeat iteratively:
   (a) Choose a leaf node t that is suitable for splitting. If there is no such node, terminate the algorithm.

(b) For node t, consider all possible splits of the node based on data from $\mathcal{D}_T(t)$. Among all possible splits, find split s_{max} that maximizes gain function

$$gain(t, s) = SSE(t) - SSE(l) - SSE(r),$$

where l and r are the nodes that would be created by splitting node t by s.

(c) Split node t using split s_{max}, creating leaves l and r. Set $c(l) = mean(y_i|(\mathbf{x}_i, y_i) \in \mathcal{D}_T(l))$, $c(r) = mean(y_i|(\mathbf{x}_i, y_i) \in \mathcal{D}_T(r))$.

The importance of choosing proper termination criteria cannot be overemphasized, as it hugely affects the performance of the resulting model. Premature termination of the iterative procedure will probably result into inability to accurately fit function f, while large trees usually overfit the data. Several approaches have been proposed to tackle this issue.

A simple practice to avoid overfitting is to shape the tree so that leaves contain relatively even number of training samples. However, it is hard to suggest any general criteria which would be robust across all applications.

Pruning

A complex approach to avoid overfitting described by Breiman et al. is based on *pruning*. Instead of striving for minimal error on the training data, we focus on finding optimal trees with respect to a cost/complexity criterion which aside from the total error on the training data also takes into account the number of leaves of the resulting tree:

$$C_\alpha(T) = SSE(T) + \alpha|\widetilde{T}|,$$

where $\alpha \in \mathbb{R}, \alpha > 0$.

Breiman et. al. show that for any of α, there exists an unique subtree $T_\alpha \subseteq T$ being optimal with respect to C_α. By choosing $\alpha = 0$, the criterion is equal to total squared error and the optimal tree is equal to the tree grown by greedy top-down recursive partitioning. On the other hand, choosing α big enough will always result in the optimal tree having only one node, as the penalty based on the number of leaves becomes the most important term in the formula.

More interestingly, it is possible to partition the interval $[0, \infty)$ into a sequence of disjoint subintervals $[\alpha_i, \alpha_{i+1})$ and construct trees $T_1 \supset \ldots \supset T_i \supset \ldots \supset T_m$ such that T_m contains only one node and for each $i = 1, \ldots, m$, T_i is optimal with respect to the cost/complexity criterion on the interval $[\alpha_i, \alpha_{i+1})$.

This sequence of subtrees can be obtained using weakest link pruning:

1. Start with the tree T_1 from top-down recursive partitioning.

2. Prune nodes iteratively until getting a one node tree. In the i-th iteration:

   (a) For each inner node $t \in T_i$, consider the prospect of collapsing it. Calculate the resulting increase of squared error $\Delta_t SSE$ and decrease in the number of leaves $\Delta_t|\widetilde{T}|$.

(b) For all inner nodes, these two measures allow us to calculate $\alpha_i(t) = \frac{\Delta_t SSE}{\Delta_t |\widetilde{T}|}$. $\alpha_i(t)$ is the border value of α, for which it is preferable to keep the subtree given by t; skipping past this value, the penalty in C_α imposed by the number of leaves outweighs better prediction given by this subtree.

(c) Select inner node $t_{min} \in T$ with minimum $\alpha_i(t)$. Collapse node t_{min}, creating tree T_{i+1}.

Having constructed the optimal pruning sequence, best subtree may be selected as the one having minimal value of the cost/complexity criterion.

Optimal pruning sequences can also be used to select best subtree using k-fold cross-validation. To do so, we construct tree T along with its optimal pruning sequence and k trees $T_1, \ldots, T_k$ used for cross-validation, also with their optimal pruning sequences. Under the assumption that the pruned subtrees of tree $T_1, \ldots, T_k$ perform similarly to subtrees of T under the same values of α, the error for a subtree of T may be calculated by averaging errors over the subtrees of $T_1, \ldots, T_k$.

Limitations

Regression trees in the presented form are piece-wise constant regression models and will probably perform poorly compared to other methods such as neural networks when approximating smooth functions.

In the construction, only one variable splits of nodes are considered and interactions between variables are omitted. Modifications of the splitting procedure exist that allow splits of the form $\sum \alpha_j X_j$, where weights α_j are optimized for the splitting criterion.

Trees are known to be highly unstable as even a small change in the data can produce different series of splits and therefore a different tree. Bagging as well as other methods presented further in this chapter are supposed to alleviate this problem.

2.1.2 Bagging

Bagging proposed by Breiman [3] is a general method for combining multiple predictors to improve prediction accuracy using the notion of *averaging*. Despite being quite simple in its nature, bagging algorithm can significatly improve performance of unstable prediction methods such as regression trees.

The Algorithm

Instead of growing a single regression tree, bagged model is obtained by growing an ensemble of regression trees $\{T_m\}_{m=1}^{M}$ independently. Each tree is grown on a data set obtained from the original training data by sampling with repetition. The algorithm:

For $m = 1, \ldots, M$:

1. Pick the m-th training set from the original training set by sampling with repetition.

2. Train regression tree T_m on the m-th training set, but do not prune.

Consecutively, the response value of the tree ensemble for point $\mathbf{x} \in dom(f)$ is obtained by averaging over the predictions given by each tree in the ensemble, hence

$$\hat{f}(\mathbf{x}) = \frac{1}{M} \sum_{m=1}^{M} T_m(\mathbf{x}).$$

A simple example of how bagging works is given in Figure 2.2.

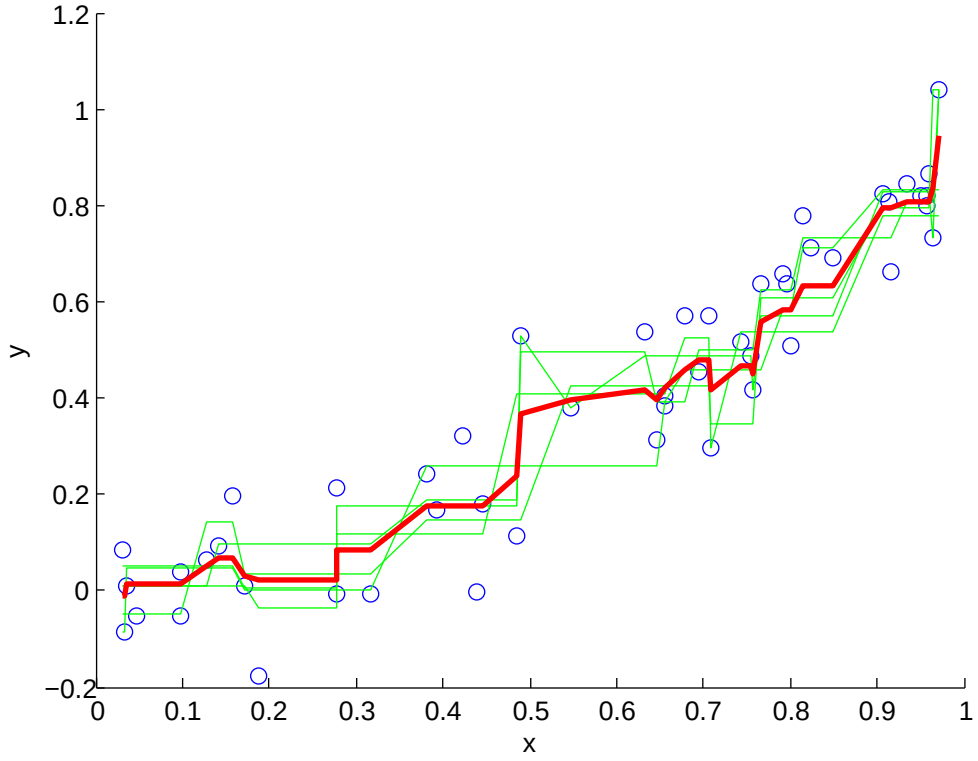


Figure 2.2: A simple example of bagging. On 50 samples obtained from a function of one variable (blue circles), 5 regression trees were grown using sampling with repetition. Responses of those five trees are shown using green lines. Red line shows the result of averaging predictions of these 5 regression trees. Despite the small number of predictors in this example, averaging already produces smoother response.

Bagging addresses several issues of regression trees. By averaging over all trees in the ensemble, the risk of overfitting is greatly reduced. Regression trees are generally not stable and even a minor change in the training set may result in a completely different shape of the tree. But with bagging, the training sets for each tree are taken with replacement, so that some samples from the original training set are present more than once in the training set for one particular tree. By growing trees without pruning, each tree gives much more accurate predictions in some parts of the sample space. Naturally, the decision to grow large trees without pruning increases the prediction variance of individual trees; however, averaging over the whole ensemble greatly reduces the total variance, improving generalization on new data.

Features

When sampling with repetition from the original training set of size N, one sample has probability $(1-\frac{1}{N})^N \approx e^{-1} \doteq 0.3679$ of not being picked into the m-th training set. As a consequence, approximately 37% of the samples are not picked into the m-th training set. These samples, usually referred to as *out-of-bag* data for tree T_m, can be later used to realistically estimate the error of the tree ensemble model. The idea is to measure performance of individual trees in the ensemble only using out-of-bag data for those particular trees and average the results. Such estimate of the generalization error is usually biased upwards. For the i-th sample in the training data, let us define by B_i the set of indices of those trees for which this sample is out-of-bag. Out-of-bag mean squared error on a training set of N training samples is then calculated as

$$MSE_{OOB}(\hat{f}) = \frac{1}{N}\sum_{i=1}^{N}\left[y_i - \frac{1}{|B_i|}\sum_{j\in B_i} T_j(\mathbf{x}_i)\right]^2.$$

The fact that bagging produces ensembles of independent trees allows us induce a distance function on an arbitrary data set $\mathcal{D}$. Having already trained an ensemble using training data, we can define δ_{ij} as the number of trees in the ensemble, for which $\mathbf{x}_i$ and $\mathbf{x}_j$ end up in the same leaf node. For an ensemble of M trees, the $M \times M$ matrix $P = \{\frac{\delta_{ij}}{M}\}_{ij}$ is often called the proximity matrix and the function $p(\mathbf{x}_i, \mathbf{x}_j) = P(i, j)$ proximity function. Distance between $\mathbf{x}_i$ and $\mathbf{x}_j$ can then be calculated as $d(\mathbf{x}_i, \mathbf{x}_j) = 1 - p(\mathbf{x}_i, \mathbf{x}_j)$.

2.1.3 Random Forests

Random forests algorithm published by Breiman [4] combines bagging with random feature selection. When splitting a node, only splits on a random subset of all variables are considered:

For $m = 1, \ldots, M$:

1. Pick the m-th training set from the original training set by sampling with repetition.

2. Train regression tree T_m on the m-th training set, using a randomly selected subset of J variables when splitting nodes (where J is constant for each iteration of the algorithm). Do not prune.

Again, the response value of the tree ensemble for point $\mathbf{x} \in dom(f)$ is calculated as

$$\hat{f}(\mathbf{x}) = \frac{1}{M}\sum_{i=1}^{M} T_m(\mathbf{x}).$$

By only considering splits on a random subset of all variables, random forests can effectively handle huge of number of variables. As in bagging, large trees are good for reducing bias, while averaging reduces variance of the predictions. Splitting on a random subset of variables gives each variable greater chance to participate when calculating ensemble response value, which especially matters

for variables that are less important and would otherwise be ignored by the regular regression tree construction procedure. Highly correlated variables also have higher chance of contributing instead of competing against each other when considering possible splits of a node.

Being an extension of bagging, random forests possess the same nice features mentioned in Section 2.1.2 such as proximity matrices and the option to calculate error based on out-of-bag data.

2.1.4 AdaBoost R2

Like bagging, boosting is also a general method of combining outputs of multiple predictors, widely recognized especially because of the success of the classification algorithm AdaBoost by Freund and Shapire ([15], [14], [33]). There is however a fundamental difference between bagging and boosting. In bagging, each predictor is trained independently of other predictors in the ensemble; in boosting, predictors are trained sequentially.

The Algorithm

Here we will present a modification of the original boosting algorithm proposed for regression by Drucker [7].

When training the first regression model, training set is picked with replacement from the original training set. Error on all samples from the original training set is then evaluated using the first regression model. Each training sample from the original training set obtains a weight, so that samples which were predicted poorly by the first regression model have bigger chance of being picked into the training set for the second regression model. This process is then repeated until obtaining an ensemble of desired size.

1. Initialize weights $w_i = 1, i = 1, \ldots, N$.
2. For $m = 1, \ldots, M$:
   (a) Pick the m-th training set from the original training set by sampling with repetition, with the i-th sample having the probability of being chosen equal to $p(i) = \frac{w_i}{\sum_{j=1}^{N} w_j}$.
   (b) Train regression tree T_m on the m-th training set.
   (c) For each sample in the training set, compute relative error as
   $$err_i = \frac{(y_i - T_m(\mathbf{x}_i))^2}{D^2},$$
   where $D = max_{j=1,\ldots,N}|y_j - T_m(\mathbf{x}_j)|$.
   (d) Calculate average error $\overline{err} = \sum err_i \cdot p_i$ and the measure of confidence $\beta_m = \frac{\overline{err}}{1-\overline{err}}$. Smaller value of β_m means higher confidence in tree T_m.
   (e) If $\overline{err} \geq 0.5$, terminate the algorithm immediately.
   (f) Change weights w_i according to the formula $w_i = w_i \cdot \beta_m^{1-err_i}$. Smaller error produces bigger decrease in w_i.

The iterative procedure returns a series of trees, each tree having assigned its measure of confidence $\{(T_m, \beta_m)\}_{m=1,\ldots,M}$.

For a particular value of $\mathbf{x} \in dom(f)$, the response value of the tree ensemble $\hat{f}(\mathbf{x})$ is calculated as a weighted median of $\{T_m(\mathbf{x})\}_{m=1,\ldots,M}$, using $ln(\frac{1}{\beta_m})$ as the weight for $T_m(\mathbf{x})$.

The convergence of the algorithms depends on the ability of regression trees to always achieve average error less than 0.5.

2.1.5 Stochastic Gradient Boosting

Gradient boosting proposed by Friedman [17] represents a way of fitting an additive model of the form of

$$\hat{f}(\mathbf{x}) = c + \sum_{s=1}^{S} \gamma_s g_s(\mathbf{x}),$$

where S is a positive integer, $c \in \mathbb{R}$, γ_s are expansion coefficients and g_s are functions picked from an arbitrary class of basis functions G. To understand how this formula relates to an ensemble of M trees, we can rewrite it as:

$$\hat{f}(\mathbf{x}) = c + \underbrace{\sum_{i=1}^{l_1} c_{i1} I(\mathbf{x} \in R_{i1})}_{T_1} + \ldots + \underbrace{\sum_{i=1}^{l_m} c_{im} I(\mathbf{x} \in R_{im})}_{T_m} + \ldots + \underbrace{\sum_{i=1}^{l_M} c_{iM} I(\mathbf{x} \in R_{iM})}_{T_M},$$

where l_m represents the number of leaves of a tree T_m, $\{c_{im}\}_{i=1,\ldots,l_m}$ are response values assigned to leaves of tree T_m and $\{R_{im}\}_{i=1,\ldots,l_m}$ are regions corresponding to leaves of this tree.

The Algorithm

Tree gradient boosting starts with $\hat{f}_0 = c, c \in \mathbb{R}$ and incrementally expands $\hat{f}$ greedily.

In the i-th iteration of the gradient boosting algorithm, steepest descent method is used to fit regression tree T_i to the negative gradient of a given loss function *Loss* by least squares, restricting the number of leaves of the tree to a small fixed number. Small fixed number of leaves is chosen due to the additive character of the expansion, where regular regression trees tend to be much too large, especially during early iterations of the algorithm.

T_i defines regions R_{ij} and response values c_j assigned to those regions. In each region, we recalculate responses values to minimize loss calculated using a given loss function *Loss*. Finally, this tree gets added to the expansion. The complete algorithm is shown in the following schema:

1. Initialize
$$\hat{f}_0(\mathbf{x}) = mean(y_i | (\mathbf{x}_i, y_i) \in \mathcal{D}_T)$$
2. For $m = 1, \ldots, M$:

(a) For $i = 1, \ldots, N$ calculate residuals

$$r_i = - \left[\frac{\partial Loss(y_i, \hat{f}_{m-1}(\mathbf{x}_i))}{\partial \hat{f}_{m-1}(\mathbf{x}_i)} \right].$$

For squared loss, residuals become

$$r_i = - \left[\frac{\partial (y_i - \hat{f}_{m-1}(\mathbf{x}_i))^2}{\partial \hat{f}_{m-1}(\mathbf{x}_i)} \right] = 2(y_i - \hat{f}_{m-1}(\mathbf{x}_i)).$$

(b) Pick $\widetilde{N} \leq N$ training samples randomly from $\{(\mathbf{x}_i, r_i)\}_{i=1}^{N}$, creating m-th training set $\mathcal{D}_T^m$. This is not a bootstrap sample, but a regular random sample picked without repetition. Using $\mathcal{D}_T^m$, construct regression tree T_m having exactly J leaves (where J is constant for all values of m).

(c) For each leaf $l_j \in T_m, j = 1, \ldots, J$, recalculate $c(l_j)$ as

$$c(l_j) = argmin_c \sum_{(\mathbf{x}, y) \in \mathcal{D}_T^m(l_j)} Loss(y, \hat{f}_{m-1}(\mathbf{x}) + c).$$

For squared loss, $c(l_j)$ will not change and this step can be skipped.

(d) Set $\hat{f}_m(\mathbf{x}) = \hat{f}_{m-1}(\mathbf{x}) + \upsilon \cdot T_m(\mathbf{x})$, where $0 < \upsilon \leq 1$ is an optional shrinkage parameter.

Friedman et. al. [19] also report that the choice of $\widetilde{N} = \frac{N}{2}$ together with $4 \leq J \leq 8$ works quite in the practice, while it is unlikely that $J > 10$ will be necessary. Smaller values of υ ($\upsilon < 0.1$) seem to dramatically increase generalization ability of the resulting model, there is however a cost to pay for it in terms of increased computational complexity as smaller values of υ often mean slower convergence and therefore larger tree ensembles.

2.2 Evolutionary Algorithms

Evolutionary algorithms belong to the family of stochastic optimization methods, being inspired by processes occurring in the nature, most notably selection, crossover and mutation. Figure 2.3 shows a schema of a typical evolutionary algorithm.

Various methods of evolutionary computation have been developed in the past decades such as *genetic programming* ([27],[26]), *evolutionary programming* ([13],[12]), *genetic algorithms* ([18],[20]) and *evolution strategies* ([34],[1]). Since *genetic programming* and *evolutionary programming* are techniques for evolving whole programs and their parameters, these techniques will not be discussed further. On the contrary, genetic algorithms and evolution strategies are quite relevant as they can be used to solve general optimization problems. In this thesis, genetic algorithms were adopted as the main optimization method.

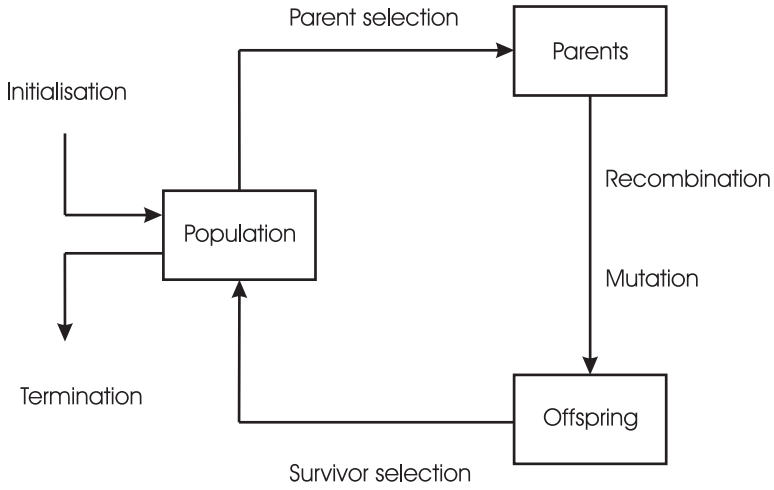


Figure 2.3: Evolution algorithm schema taken from the book on evolutionary computing by Eiben and Smith [8].

Evolution algorithms are quite popular as global optimizers due to the fact that they do not require any assumptions to be made about the characteristics of the optimized function. Instead, evolution algorithms work directly with the values sampled from the underlying distribution. This feature allows the evolutionary techniques to be applied in optimization of the so-called *black-box* or *empirical* functions, where the analytical description of the optimized function is not available or does not even exist at all.

However, this strength of evolutionary algorithms comes at a price - compared to specialized optimization methods which make assumptions about the objective function (eg. gradient methods used for smooth functions), evolutionary algorithms converge rather slowly and require a large number of function samples to be evaluated.

2.2.1 Genetic Algorithms

In genetic algorithms ([18],[20]), candidate solutions of an optimization problem $minf(\mathbf{x})$ are thought of as *individuals*. Each *individual* is represented by its

genetic code, which may be in the simplest case a binary vector of a fixed length or some arbitrarily complex problem-dependent representation of the domain of the optimized function.

For f being a function of n categorical and continuous variables, representation via binary strings is often impractical due to issues concerning the way numbers are stored in the memory. Instead of binary strings, we will use a more high-level representation: for $\mathbf{x} \in dom(f)$, we will encode individual ind_x simply as a vector of values of individual variables $ind_x = (x_1, \ldots, x_n)$.

In the nature, the quality of an individual is indirectly measured by its ability to survive and reproduce in a hostile environment. In genetic algorithms, the pressure of the environment is simulated by defining for each individual its measure of quality called *fitness*. Fitness usually takes the form of a function that assigns to each individual a real number, bigger number standing for a better individual. For the minimization problem $minf(\mathbf{x})$, the definition of fitness can be quite straightforward, for example $F(ind_x) = -f(\mathbf{x})$.

Population of individuals represents a set of problem solutions of various quality (ie. with different values of fitness). Genetic algorithm directly ties the value of fitness to the ability of that individual to reproduce. In the reproduction step, genetic code of the parent individuals is recombined using crossover operators to create offspring individuals. Genes of the offspring individuals may not be dependent entirely on the genetic information from the parent population, as mutations may occur to add diversity to the offspring.

Finally, new population is formed by picking individuals from both parent and offspring populations; the whole process is then repeated with the obtained new population, as shown in Algorithm 2.1.

Algorithm 2.1 An outline of a simple genetic algorithm.

$t \leftarrow 1$ {initialize the number of iterations}
$P_1 \leftarrow$ select initial population

while $t < t_{max}$ **do**
  $F_t \leftarrow EvaluateFitness(P_t)$
  $O_t \leftarrow SelectAndRecombine(P_t, F_t)$
  $P_{t+1} \leftarrow CreateNewPopulation(P_t, O_t)$

  $t \leftarrow t + 1$
end while

output $\leftarrow$ best solution from $\bigcup P_i$

Selection

Selection is responsible for focusing the search on those areas of the search space with high values of fitness.

To describe selection mechanisms, we will assume that the range of the fitness function are positive real numbers, higher number standing for better solution.

Probably the simplest selection method is *proportionate* (sometimes called *roulette-wheel*) selection. For a population of n individuals $P = \{ind_i\}_{i=1}^{n}$, total

fitness across the whole population is calculated as $F = \sum_{i=1}^{n} F(ind_i)$. *Proportionate* selection assigns to each individual its probability of being picked $p(ind_i) = \frac{F(ind_i)}{F}$ and consequently samples from the population using this probability distribution.

Stochastic universal selection is a development of the *proportionate* selection mechanism, where instead of making N independent draws with repetition, all N individuals are selected at once. The population is sorted in ascending order by fitness values, let $ind_{j_1}, \ldots, ind_{j_{|P|}}$ be the resulting ordering and let us define cumulative fitness for individual ind_{j_k} as $F'_k = \sum_{l=1}^{k} F(ind_{j_l})$. Definition of cumulative fitness F' allows us to construct intervals $I_1 = [0, F'_1), I_2 = [F'_1, F'_2), \ldots, I_{|P|} = [F'_{|P|-1}, F'_{|P|})$. The interval $[0, F) = I_1 \cup \ldots \cup I_{|P|}$ is sampled in N points $\{r + i\frac{F}{N}\}_{i=0}^{N-1}$ for some $r \in [0, \frac{F}{N})$ chosen randomly. Due to the correspondence between ind_{j_k} and I_k, these N points then exactly determine N individuals to be picked from the parent population.

Another popular type of selection is the *tournament* selection with integer parameter $t > 1$, which involves the concept of running tournaments in subpopulations of size t picked from the original population randomly (typically by sampling with repetition from the uniform distribution). For each tournament of t individuals, the winner is usually chosen to be the individual with the highest fitness from those t individuals participating in the tournament. Selection pressure can be adjusted by changing tournament size parameter t. Smaller values of t result in an increased chance of being chosen even for weak individuals, thus maintaining diversity in the offspring population.

Some selection techniques rescale fitness values as a mean to avoid preliminary convergence of the genetic algorithm. An examples of such approach is the *rank* selection, which ranks the population first and consecutively substitutes ranks for the fitness function values.

Crossover and Mutation Operators

Crossover and mutation operators serve as the means of exploring the search space.

Crossover in its simplest form is carried out by *one-point* and *two-point* crossover operators (depicted in Figure 2.4), which interchange continuous segments of the genetic code between individuals. *Uniform* crossover is a similar approach that chooses cut points to be all boundaries between genes (ie. larger functional units of the genetic code), consequently interchanging approximately one half of genes between individuals.

The forementioned crossover types work well for individuals represented with binary strings; for problem-dependent representations (such as the vector of values of individual variables), these operators only interchange values of individual variables, which may lead to decreased population diversity and ultimately to unsatisfactory exploration of the search space. Such drawback may be compensated either by extensive use of mutations or by applying special types of crossover operators. An example of such specialized type of a crossover operator for vectors of real numbers is the *arithmetic* crossover, which does not interchanges variable values, but rather takes their linear combination.

Mutation of an individual represented by a binary string is usually performed

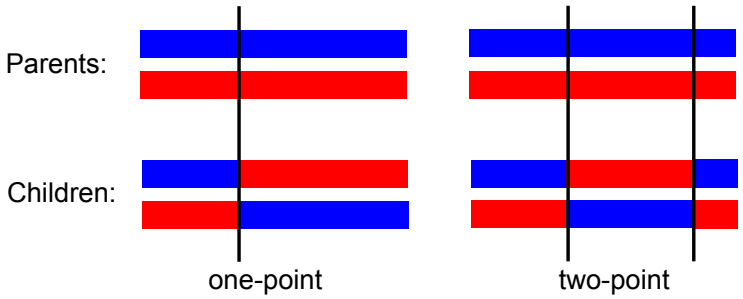


Figure 2.4: Visualization of one-point and two-point crossover operators.

by changing value of a single bit. For representation by vectors of variable values, mutation changes value of a single variable to a different value. For continuous variables, this may be done by adding a random number from the uniform distribution $U(-a, a)$ or from the normal distribution $\mathcal{N}(0, \sigma^2)$.

Creating new population

Various strategies may be utilized for creating new population from parent and offspring populations. One option is to create the new population only from the newly created offspring individuals. There is however no guarantee of any kind that the fitness will grow across generations; in fact, the direct opposite may happen.

To counter this problem, parent and offspring populations may be combined so that the fittest individuals from the parent population survive in the new population - this approach is called *elitism*. Under *elitism*, few chosen individuals with the highest value of fitness in the present population are put directly into the newly created population, skipping crossover and mutation operators.

The actual form and order of applying genetic operations may differ across implementations. For example, the offspring population may be split into parts, individuals in each part being created by one particular genetic operator.

Termination criteria

The main optimization loop consisting of applying genetic operators terminates upon reaching certain termination criteria.

These criteria may be either a fixed allowed number of generations or convergence of the algorithm measured by negligible gains in the best values of fitness in the past few generations. For empirical objective functions, a common criterion is exhaustion of a given computational budget, expressed as the number of objective function evaluations.

2.2.2 Optimization with Constraints

Under many circumstances, domain of the function f for which we run the optimization algorithm may be constrained. For example, only certain combinations of categorical variables may be allowed or, conversely, prohibited. Linear or nonlinear constraints may be applied to combinations of continuous variables.

In this thesis, only linear constraints will be employed. For a vector of continuous variables $\mathbf{x} = (x_1, \ldots, x_n)^T$, linear constraints are defined by a finite set of m^{ineq} inequality and m^{eq} equality linear constraints. We can formulate the constraints using matrix notation as

$$\mathbf{A}^{ineq}\mathbf{x} \leq \mathbf{b}^{ineq}$$

and

$$\mathbf{A}^{eq}\mathbf{x} = \mathbf{b}^{eq}$$

for some $\mathbf{A}^{ineq} \in \mathbb{R}^{m^{ineq} \times n}$, $\mathbf{A}^{eq} \in \mathbb{R}^{m^{eq} \times n}$, $\mathbf{b}^{ineq} \in \mathbb{R}^{m^{ineq} \times 1}$ and $\mathbf{b}^{eq} \in \mathbb{R}^{m^{eq} \times 1}$.

It is usually reasonable to assume that the polytopes defined by these linear constraints are bounded.

Basically, there are two ways to work with linear constraints in an evolutionary algorithm. We can allow infeasible solutions to exist in the population and modify the calculation of fitness in such a way so that it penalizes these solutions to certain extent. Such solution is simple and usually effective, but in some cases, evolutionary algorithm may not be able to find any feasible solutions.

Another option is to modify crossover and mutation operators to consider these linear constraints and always produce only feasible solutions (this is also the path taken in this thesis).

2.3 Surrogate Modelling

As already mentioned in the introduction of the thesis, evaluation of an empirical objective function may require running costly experiments or simulations. For example, in airfoil shape optimization ([39]), air flow around the wing must be simulated based on input design variables; such simulation may takes hours or even days. Since evolutionary optimization algorithms typically require a large number of samples to be evaluated from the domain of the objective function, the fact that the objective function is empirical means serious complications as the evaluation of fitness becomes slow and costly. Instead of always evaluating the objective function, approximate regression model of the objective function may be constructed; such *surrogate model* can later be used in place of the empirical objective function to reduce the number of objective function evaluations.

In evolutionary optimization, researchers have typically utilized non-linear regression methods to construct surrogate models, including Gaussian processes ([9], [5], [39]), neural networks ([21], [39]) and support vector machines ([35]). An overview survey of fitness approximation techniques for evolutionary optimization was published by Jin [22].

Evolution control is a widespread term designating the method by which the surrogate model is employed in the optimization algorithm. Two basic approaches exist to evolution control - individual-based and generation-based control.

Having already obtained a sufficiently large set of empirical objective function samples, approximate model may be constructed before the first iteration of the optimization procedure. If this is not the case, optimization algorithm is usually run for some fixed number of iterations using exclusively the empirical objective function to gather initial training data for the model.

2.3.1 Individual-based control

In each generation, individual-based evolution control ([23], [6]) evaluates certain number of selected individuals using the objective function, remaining individuals are evaluated using the surrogate model. There exist various strategies for picking *controlled* individuals, which will be evaluated using the objective function ([22],[24]):

- **Best strategy** - in this strategy, only best individuals (measured by the approximate model) are reevaluated using the objective function.
- **Random strategy** - using the objective function, random strategy reevaluates a fraction of individuals picked randomly from the current population.
- **Worst prediction error strategy** - in this strategy, those individuals are reevaluated by the objective function which are approximated poorly by the surrogate model. The quality of approximation is measured by estimated prediction error of the surrogate model.
- **Clustering strategy** - after the individuals are grouped into clusters by a given clustering algorithm, one (or more) points from each cluster are evaluated using the objective function. These points may be chosen in many ways, eg. randomly, as points closest to centers of the clusters or points having best fitness within each cluster (where the fitness is evaluated using the surrogate model).

2.3.2 Generation-based control

Generation based control ([23], [32]) employs the objective function only for selected generations (called *controlled* generations), remaining generations are evaluated by the approximate model. Controlled generations may be selected heuristically (typically by specifying that in a cycle of g generations, first $g_m < g$ generations are evaluated using the surrogate model and the remaining generations using the objective function), based on the quality of the approximate model or depending on the convergence of the optimization algorithm on the approximate model.

2.3.3 More Examples of Evolution Control Methods

Emmerich et. al. [10] and Ulmer et. al. [36] employ individual-based control, putting to use the concept of *pre-offspring*. When generating offspring by genetic operators, a large pre-offspring population is generated first and evaluated using the approximate model. Best individuals from the pre-offspring are then reevaluated using the objective function and picked into the next population.

Zhou et. al. [39] propose an approach that utilizes Gaussian processes as a global and radial basis networks as a local model in airfoil shape optimization. In each generation, having evaluated all individuals by the global model, local model together with a gradient-based search method is used to improve fitness of the best individuals identified by the global model.

2.4 Evolution in Catalysis

Evolutionary optimization has been successfully employed in catalysis ([30],[37]) to search for optimal mixtures of catalyst elements, which is a typical problem of optimizing an empirical objective function due to the fact that the properties of the obtained catalysts must be assessed experimentally. Common formulation of an optimization problem in catalysis has the following characteristics:

- **Probability distribution on discrete solutions** - each feasible combination of values of the discrete variables $\mathbf{x}^d$ may have assigned its probability $p(\mathbf{x}^d)$. The optimization algorithm should prefer to search areas of the domain of the objective function with high values of p.
- **Hierarchy of variables** - discrete variables may be organized into hierarchies. For example, we can have groups and subgroups of catalyst elements and decide which groups and subgroups of elements will be used in the catalyst and which will be not.
- **Presence of count variables** - number of elements chosen from a group or a subgroup of elements to form the catalyst is often determined by a discrete integer variable $x^d \geq 0$.
- **Dependence of continuous variables on categorical variables** - each feasible combination of discrete variables specifies the qualitative composition of the catalyst, ie. which elements and compounds will be used. Some of the elements or compounds may have assigned quantitative (continuous) variables that specify proportions in which the element or compound should be used.
- **Linear constraints** - numeric variables are constrained by linear constraints in the matrix form $\mathbf{A}^{ineq}\mathbf{x} \leq \mathbf{b}^{ineq}$ and $\mathbf{A}^{eq}\mathbf{x} = \mathbf{b}^{eq}$. The matrix formulation of the linear constraints applies to all solutions; in the case when some numeric variables are not relevant for one particular solution, they are considered to be zero and the corresponding columns may be dropped from $\mathbf{A}^{ineq}$ and $\mathbf{A}^{eq}$.

A nice example of a hierarchical catalyst structure is shown in Figure 2.5.

2.4.1 Preprocessed Form of the Optimization Problem

As previously mentioned, each discrete solution defines a polytope on some of the continuous variables through linear constraints. The number of discrete solutions for one particular problem may be huge (in order of billions), the number of non-empty polytopes, however, may be relatively small. Moreover, huge numbers of obtained polytopes are often isomorphic and can be transformed into each other by permuting rows and columns of linear constraint submatrices. Therefore, we can save many costly polytope computations by aggregating discrete solutions into equivalence classes based on isomorphism (in the the aforementioned sense) of the polytopes and propose the genetic algorithm for such representation.

Each polytope equivalence class will thus contain:

- Bounded polytope represented by a set of inequality constraints $\mathbf{Ax} \leq \mathbf{b}$, derived from original inequality and equality linear constraints.
- Set of discrete solutions (with assigned probabilities), represented as a Cartesian product of values of discrete variables.
- Isomorphism between each discrete solution and the polytope given by $\mathbf{A}$ and $\mathbf{b}$.

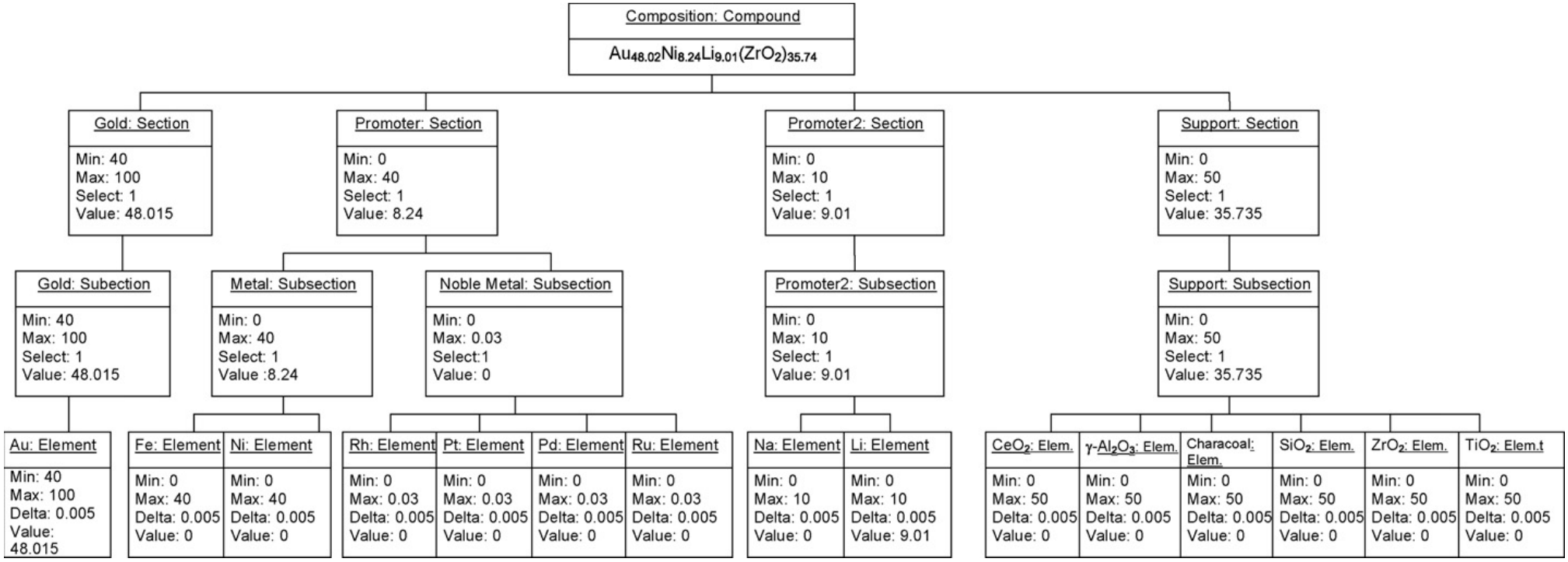


Figure 2.5: Example of a catalyst structure taken from [37]. Hierarchical dependence between variables is illustrated by sections and subsections of catalyst elements. On the bottom level, there are 5 groups of elements, only 1 element being picked from each group (as specified by the `Select` property). Proportions of each element have assigned its lower bound `Min` and upper bound `Max`.

3. Proposed Genetic Algorithm

In this chapter, genetic algorithm is proposed to solve optimization problems encountered in catalysis (as described in Section 2.4), utilizing a surrogate model based on regression trees and their ensembles.

3.1 Genetic Algorithm Outline

The main part of the optimization algorithm is presented as Algorithm 3.1, with the process of fitness evaluation presented separately in Algorithm 3.2.

In the outline of the algorithm, p denotes the size of the population (p is constant for all generations). Stopping criteria for the algorithm are represented by t_{max} and e_{max} parameters, where t_{max} is the maximum number of iterations of the genetic algorithm and e_{max} is the computational budget, ie. the allowed number of objective function evaluations. Minimum size of the database `DB` required to train the model has to be specified in parameter s_{min}.

The proposed algorithm supports both individual-based and generation-based evolution control. With individual-based control, in each generation $k \cdot p$ individuals are evaluated using the surrogate model and p individuals are consecutively reevaluated by the surrogate model. Parameter g specifies the length of the cycle of generation-based control, in which first $g_m < g$ generations are evaluated using the model and the remaining generations using the objective function.

3.2 Fitness Scaling

The proposed genetic algorithm is restricted to non-negative fitness values. Due to the fact that we are solving a minimization problem, fitness is computed by negating objective function values (or by negating predictions from the surrogate model). We propose the following scaling operators:

- **Simple scaling** - simple scaling is designed to tackle negative fitness values, when fitness is non-negative for all individuals in the population, simple scaling leaves the values unchanged. In presence of negative fitness values, simple scaling calculates minimal fitness value F_{min} and subtracts it from all fitness values. To avoid creating individuals with zero fitness, individuals which previously had fitness F_{min} get assigned the second lowest fitness in the population.

- **Linear scaling** - in linear scaling, user specifies parameters $a, b \in \mathbb{R}$ and the fitness for each individual is recomputed as $aF(ind) + b$.

- **Rank scaling** - rank scaling recomputes fitness based on the ranking of individuals, where the fittest individual is ranked 1 and the worst individual n. Operator is parametrized by a function $f(r)$ that based on the rank calculates scaled fitness, on default $f(r) = \frac{1}{\sqrt{r}}$. An example of rank scaling is given in Figure 3.1.

Algorithm 3.1 An outline of the proposed genetic algorithm.

$t \leftarrow 1$ {initialize the number of iterations}
$e \leftarrow 0$ {initialize the number of objective function evaluations}

if initial population is not specified **then**
  $P_1 \leftarrow$ random initial population
  $F_1 \leftarrow$ fitness evaluated using the objective function
end if

if $|\texttt{DB}| \geq s_{min}$ **then**
  `model` $\leftarrow$ train model on data from `DB`
end if

while $t < t_{max}$ and $e < e_{max}$ **do**
  $F'_t \leftarrow$ (optionally) rescale fitness F_t

  if (individual-based evolution control) and (`model` is available) **then**
    $o \leftarrow k \cdot p$
  else
    $o \leftarrow p$
  end if

  Using selection operator on (P_t, F'_t):
  $O_t \leftarrow$ create o offspring individuals using genetic operators

  $F \leftarrow$ EvaluateFitness(O_t,`DB`,t,e,k,g,g_m,s_{min}) {Algorithm 3.2}

  $O^e_t \leftarrow$ pick l fittest individuals from (P_t, F'_t) {Elitism}

  if (individual-based evolution control) **then**
    $O'_t \leftarrow$ those individuals from O_t evaluated using the objective function
    $P_{t+1} \leftarrow O^e_t \cup O'_t$
  else
    $P_{t+1} \leftarrow O^e_t \cup O_t$
  end if
  $F_{t+1} \leftarrow$ fitness of individuals selected into P_{t+1}

  $t \leftarrow t + 1$
end while

output $\leftarrow$ best solution from `DB`

Algorithm 3.2 Fitness evaluation in the proposed genetic algorithm.

```
if |DB| < s_min then
  F ← fitness of O_t evaluated using the objective function
else if (individual-based evolution control) then
  F ← fitness of O_t evaluated using the model
  O'_t ← select p individuals from O_t for revaluation by the objective function
  F ← reevaluate individuals from O'_t using the objective function
  e ← e + p
else if (generation-based evolution control) then
  if (in the cycle of g iterations, the model was already used in g_m iterations)
  then
    F ← fitness of O_t evaluated using the objective function
    e ← e + p
  else
    F ← fitness of O_t evaluated using the model
  end if
end if

if data in DB have changed then
  model ← train model on data from DB
end if
```

3.3 Selection

Although the proposed algorithm supports arbitrary selection operators, for the purpose of prototype implementation and consequent testing, only operators described in section 2.2.1 (*proportionate*, *tournament* and *stochastic uniform* selection) were considered.

Selection operator expects all crossover operators to create one offspring individual from two parents; this way, the number of selected individuals can be calculated as $2p_c + p_m$, where p_c and p_m are numbers of offspring individuals created by crossover and mutation.

3.4 Genetic Operators

The proposed genetic algorithm supports multiple crossover and mutation operator types. Each genetic operator instance has assigned a real number $0 < f \leq 1$, so that when generating o offspring individuals in one iteration of the optimization algorithm, $o \cdot f$ individuals are generated using this operator instance.

3.4.1 Crossover Combined with Mutation

For the specific type of constrained optimization problems encountered in catalysis, we propose one specialized crossover operator which combines crossover with random mutation (meaning that in complicated cases, part of the offspring individual is chosen randomly). Let us assume that we want to crossover two solutions

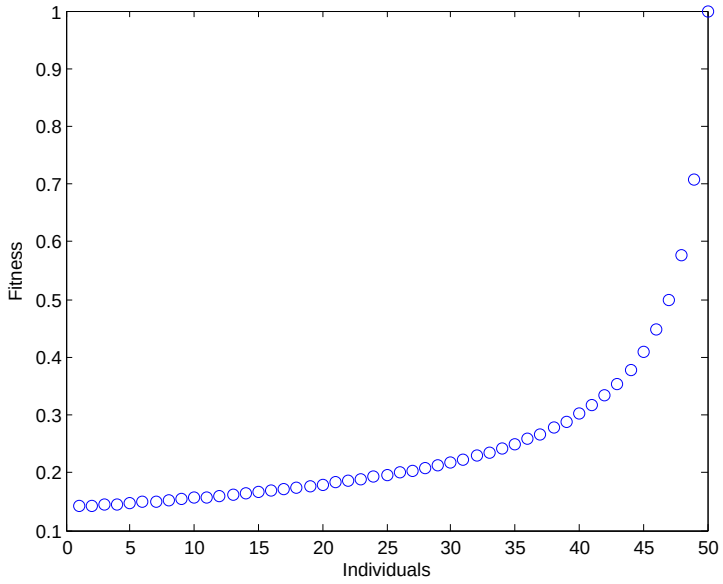


Figure 3.1: Rank scaling example. The picture shows rank-scaled fitness of 50 individuals with $f(r) = \frac{1}{\sqrt{r}}$. For demonstration purposes, individuals are arranged in ascending order by their fitness.

$\mathbf{x} = (\mathbf{x}^d, \mathbf{x}^c)$ and $\mathbf{y} = (\mathbf{y}^d, \mathbf{y}^c)$ to a problem having both discrete and continuous variables.

The operator handles crossover of values of categorical and continuous variables separately, starting with the values of the categorical variables. When constructing offspring individual $\mathbf{s} = (\mathbf{s}^d, \mathbf{s}^c)$, it first looks for possibility of interchanging values of categorical variables, considering i discrete solutions of the form $\mathbf{s}_i^d = (x_1^d, \ldots, x_{i-1}^d, y_i^d, x_{i+1}^d, \ldots, x_n^d), x_i^d \neq y_i^d$, where n is the number of categorical variables. When none of $\mathbf{s}_i^d$ is feasible, the algorithm switches $\mathbf{x}$ and $\mathbf{y}$ and repeats this step.

If none feasible solutions are found even then, $\mathbf{s}^d$ is chosen from the set of all discrete solutions as the solution that minimizes the sum of squared Hamming distances $d(\mathbf{x}^d, \mathbf{y}^d, \mathbf{s}^d) = h(\mathbf{x}^d, \mathbf{s}^d)^2 + h(\mathbf{y}^d, \mathbf{s}^d)^2$, where $h(\mathbf{u}^d, \mathbf{v}^d) = \frac{\sum_{i=1}^n I(u_i^d \neq v_i^d)}{n}$ is the Hamming distance.

On the other hand, having found multiple feasible discrete solutions $\mathbf{s}_i^d$, one of them is chosen randomly and denoted $\mathbf{s}^d$. The crossover of values of discrete variables then stops with a given probability p or continues in the same fashion iteratively, so that in the j-th iteration, $j > 2$, values of another $j - 1$ variables are interchanged between $\mathbf{s}^d$ and $\mathbf{y}^d$ and there is a chance of p^j that the algorithm will stop at the end of that particular iteration.

Notably, in the case when $\mathbf{x}^d$ and $\mathbf{y}^d$ come from the same polytope equivalence class, $\mathbf{s}^d$ also belongs to this polytope equivalence class - this is a direct consequence of the fact that we represent discrete solutions corresponding to a polytope equivalence class by a Cartesian product of values of discrete variables.

To decide values of continuous variables, arithmetic crossover is performed on values of continuous variables from $\mathbf{x}$ and $\mathbf{y}$ if the situation allows it, which is when both $\mathbf{x}^d$, $\mathbf{y}^d$ and $\mathbf{s}^d$ represent solutions from the same polytope equivalence class. When the polytope equivalence class of $\mathbf{s}^d$ is the same as that of $\mathbf{x}^d$ ($\mathbf{y}^d$), but not as that of $\mathbf{y}^d$ ($\mathbf{x}^d$), $\mathbf{s}^c = \mathbf{x}^c$ ($\mathbf{s}^c = \mathbf{y}^c$). If none of the previous cases apply, values of continuous variables are chosen randomly from the potytope corresponding to

$\mathbf{s}^d$.

3.4.2 Mutation

For constrained mixed optimization problems, we propose three mutation operator types for continuous variables and one operator type for both discrete and continuous variables:

- **Mutation on continuous variables (Step)** - in this type of mutation, some continuous variables are changed by a small random step $x_i = x_i + \epsilon_i$. Each continuous variable has a chance $0 < p_r \leq 1$ of being chosen for mutation, meaning that all or none continuous variables may be changed by the mutation operator. Upon deciding which of the continuous variables will be adjusted, these variables are permuted and changed in the obtained order. Value of a mutated continuous variable x_i is adjusted by random ϵ_i smaller than a specified maximum step size Δ chosen from the uniform distribution, so that $x_i = x_i + \epsilon_i$ satisfies the linear constraints. For degenerate cases, when linear constraints define subspace of lower dimension than the space on which they are defined, this type of mutation will most likely not work. In such case, projection of the polytope to lower-dimensional subspace is neccessary.

- **Mutation on continuous variables (Ball)** - this mutation is similar to the previous one, except for the fact that this time, all continuous variables are mutated at once, so that for a vector of continuous variables $\mathbf{x}^c = \mathbf{x}^c + \epsilon$, where ϵ is a random point from a ball of radius Δ. Again, this mutation will not work for degenerate polytopes, where projection into a lower-dimensional subspace is necessary.

- **Mutation on continuous variables (Cones)** - third continuous mutation implements the approach based on tangent cones proposed by Torczon and Lewis in [29] and [28]. The key idea is that for a given $\mathbf{x}^c$ and $\epsilon \in \mathbb{R}, \epsilon > 0$, *active* linear constraints are identified first - those are the contraints represented by hyperplanes intersecting the ball of origin $\mathbf{x}^c$ and radius ϵ. Let us assume that we have r active contraints with normals $\{\mathbf{a}_i\}_{i=1}^r$. These r normals define finitely generated cone $N(\mathbf{x}, \epsilon) = \{\lambda_1\mathbf{a}_1 + \ldots + \lambda_r\mathbf{a}_r | \lambda_i \in \mathbb{R}, \lambda_i \geq 0\}$. Points from the set $\mathbf{x} + T(\mathbf{x}, \epsilon)$ given by the tangent cone $T(\mathbf{x}, \epsilon) = \{\mathbf{y} | \mathbf{v} \cdot \mathbf{y} \leq 0 \text{ for } \forall \mathbf{v} \in N(\mathbf{x}, \epsilon)\}$ are guaranteed to satisfy all linear constraints in the neighborhood of $\mathbf{x}$ (see Figure 3.2 for an example). Generating vectors of tangent cone $T(\mathbf{x}, \epsilon)$ can be found among the columns of matrices $V(V^TV)^{-1}, -V(V^TV)^{-1}, U$ and $-U$, where V denotes the matrix whose columns are generators of $N(\mathbf{x}, \epsilon)$ and columns of U form the basis of nullspace of V^T. Generating vectors can be subsequently used as step directions; such approach is likely to work even in degenerate cases.

- **Mutation on all variables** - each discrete solution $\mathbf{x}^d$ obtained by choosing values of n categorical variables has assigned its probability $p(\mathbf{x}^d)$ of being explored by the evolutionary algorithm. For two discrete solutions $\mathbf{x}^d$ and $\mathbf{y}^d$, we can calculate the Hamming distance as $h(\mathbf{x}^d, \mathbf{y}^d) = \frac{\sum_{i=1}^n I(x_i^d \neq y_i^d)}{n}$.

The proposed mutation for a given $\mathbf{x}$ calculates for all feasible $\mathbf{y}^d \neq \mathbf{x}^d$ weights $w(\mathbf{y}^d)$ as $w(\mathbf{y}^d) = [1 - h(\mathbf{x}^d, \mathbf{y}^d)] \cdot p(\mathbf{y}^d)$ and then chooses the new discrete solution randomly by sampling from the probability distribution given by $w(\mathbf{y}^d)$. If the new solution belongs to the same polytope equivalence class, the values of the continuous variables may remain the same. Otherwise, values of continuous variables are chosen randomly within the polytope defined by $\mathbf{y}^d$. The mutation allows to explicitly restrict the set of considered discrete solutions only to those solutions $\mathbf{y}^d$ contained in the same polytope equivalence class as $\mathbf{x}^d$ or to those solutions with $h(\mathbf{x}^d, \mathbf{y}^d) = 1$.

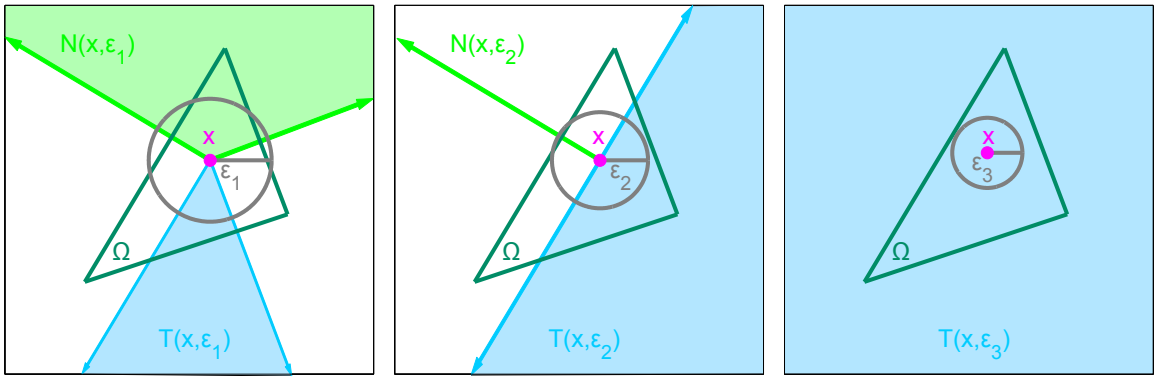


Figure 3.2: Tangent cones example (taken from [28]). The picture shows normal and tangent cones for three different values of ϵ on a two-dimensional polytope $\Omega \subset \mathbb{R}^2$. In the third case, $N(\mathbf{x}, \epsilon_3) = \{\mathbf{0}\}$ and $T(\mathbf{x}, \epsilon_3) = \mathbb{R}^2$ as $\mathbf{x}$ lies far from the polytope boundary.

3.5 Model Training and Selection

To the surrogate model, data samples are presented as a vector $(\mathbf{x}^d, \mathbf{x}^c, \mathbf{b})$, where $\mathbf{x}^d$ and $\mathbf{x}^c$ are discrete and continuous parts of one particular solution and $\mathbf{b}$ is a bit vector for which $b_i = 1$ if and only if the continuous variable x_i^c is active in this particular solution. Special value is reserved for each discrete variable for the case when this variable is not active in the solution. Continuous variables which are not active are considered to be equal to zero.

Given regression trees, four different types of regression tree ensembles and plenty of parameter choices, it is hard to decide which model to choose and how exactly to set up the parameters for the given black-box optimization problem. Therefore, we propose two straightforward ways of addressing this issue:

- **Single model only** - given a black-box optimization problem, it is the user who decides which model should be used, how to train the model and how to set up the parameters. The selected model is then used during the whole optimization process.

- **Tournament of models** - given a black-box optimization problem and a set of models specified by the user, pick the best model for the given

situation. The user may choose if the best surrogate model is decided only once upon obtaining enough data samples or if the choice is made each time more data samples get evaluated using the objective function. To select the best model, available data samples evaluated with the objective function are divided randomly between a training set and a test set in a given ratio. All model types are trained on the training data and the model with the best mean squared error on the test data is selected as the surrogate model. Finally, the best surrogate model is retrained using all data samples.

3.6 Random Initial Population

In the case when initial population is left unspecified, the algorithm generates random population in two steps. In the first step, p feasible discrete solutions $\{\mathbf{x}_i^d\}_{i=1}^p$ are chosen randomly by sampling from the probability distribution on all discrete solutions.

For each discrete solution $\mathbf{x}_i^d$, continuous part of the solution is computed as a random point in the polytope corresponding to $\mathbf{x}_i^d$. There exist many algorithms for random sampling from polytopes, ranging from simple (such as random sampling from the polytope's bounding box) to more complicated ones based eg. on random walks ([38],[25]). We will use two algorithms to obtain a random point inside the polytope:

- **Ball random walk** - given an arbitrary inner point of the polytope, this point is moved n times randomly to its neighborhood given by a ball of a specified radius. The point obtained after these n steps is returned as the random point. If the polytope is degenerated (ie. does not have the same dimension as the space on which it is defined), it must be projected into a subspace of lower dimension for this algorithm to work.

- **Random walk using generating vectors of tangent cones** - given an arbitrary inner point of the polytope, this point is moved n times randomly to its neighborhood using generators of tangent cones similarly to **Mutation on continuous variables (3)** proposed in section 3.4.2. The point obtained after these n steps is returned as the random point.

3.7 Individual-based Evolution Control

In individual-based control, decision must be made in each generation about which offspring individuals will be evaluated using the objective function and which using the surrogate model (examples of individual-based strategies are given in section 2.3).

In the proposed optimization algorithm, combination of multiple strategies is possible, each strategy responsible for selecting only a fraction of the total of p individuals for revaluation by the objective function.

Strategies are applied in a given order, each consecutive strategy being aware of choices made by previous strategies, so it can choose the individuals to be reevaluated from the set of individuals which have not been picked by previous strategies. As a result, strategies for picking individuals for reevaluation using

the objective function can cooperate. Following strategies were chosen for implementation:

- **Best strategy** - in the best strategy, specified fraction of individuals having the best fitness according to the surrogate model is reevaluated using the objective function.
- **Random strategy** - in the random strategy, specified fraction of individuals is chosen randomly for revaluation with the objective function.
- **Random weighted strategy** - in random weighted strategy, specified fraction of individuals is chosen randomly for revaluation with the objective function, but this time, each sample is assigned a weight defining its probability of being picked for revaluation. The weight is assigned to each sample using the proximity matrix established by either bagging or random forests (see section 2.1.2 for details). Weight for each individual is calculated as inverse of the sum of squared proximities to other individuals $w(\mathbf{x}_j) = \frac{1}{\sum_{i=1}^{n} p(\mathbf{x}_i, \mathbf{x}_j)^2}$, where $p(\mathbf{x}_i, \mathbf{x}_j)$ is defined as in section 2.1.2.
- **Clustering strategy** - in clustering strategy, individuals are grouped into c clusters by agglomerative hierarchical clustering, where c is the number of individuals which should be chosen for reevaluation by this strategy. In the clustering algorithm itself, distance is measured using the proximity matrix established by either bagging or random forests (see section 2.1.2 for details). Point with the best fitness according to the surrogate model is chosen from each cluster for revaluation by the objective function.

4. Implementation

Implementation of the proposed methods was carried out in *MATLAB 2010b*, an environment developed by *Mathworks* for technical computations. *MATLAB* offers great extensibility in the form of toolboxes that focus on different areas of technical computing. For the purposes and needs of this thesis, *MATLAB* was used along with *Statistics Toolbox*[1] that provides neccessary statistical tools and also an implementation of regression trees, and *Multi-Parametric Toolbox*[2], which contains useful polytope routines.

Prototype implementation of the proposed methods is supplied on an attached CD. User documentation is present in the form of demonstration scripts showing typical usage of the prototype implementation in various scenarios.

In the following sections, we will review in detail the most important parts of the prototype implementation as well as encountered problems and explain reasons for some of the decisions taken in the implementation process.

4.1 Tree Ensembles

In *Statistics Toolbox*, regression trees are implemented in the `classregtree` class. For our purposes, two important existing features of `classregtree` are its support for splits based on random subsample of variables and robust implementation of bottom-up pruning based on cross-validation or a separate validation set. `classregtree` supports two stopping criteria for top-down recursive partitioning, `minleaf` and `minparent`. `minleaf` specifies the minimum allowed number of training samples in the leaves of the tree, while `minparent` stands for the minimum required number of training samples contained in parent nodes before they are split. Default values of these two settings are `minparent=10` and `minleaf=1`, for full grown trees (which are often desired in bagging and random forests) it may be preferred to use `minparent=2` and `minleaf=1`.

Statistics Toolbox also provides an implementation of bagging and random forests in the `TreeBagger` class. Unfortunately, this implementation targets only those types of tree ensembles where response values are calculated by weighted averaging over the trees in the ensemble and due to complicated source code, it would be hard to change the existing implementation to support AdaBoost R2 or stochastic gradient boosting methods. Moreover, `TreeBagger` tree ensembles do not support learning with early stopping rules.

As a consequence, a slightly different approach to tree ensembles was implemented in the base class `TreeEnsemble` and its successors `TreeBagging`, `RandomForest`, `TreeAdaBoostR2` and `TreeStochasticGradientBoosting`. Class `TreeEnsemble` defines an unified interface to tree ensembles via a set of properties and abstract methods.

Each tree ensemble is required to implement a standard set of learning methods. Let us first describe the parameters passed into these methods: `Xt` and `Yt`

[1] *Statistics Toolbox* is one of many *MATLAB* toolboxes developed by *Mathworks*

[2] *Multi-Parametrix Toolbox* is available at `http://control.ee.ethz.ch/~mpt/` under GNU general public licence

designate the training data, while `Xv` and `Yv` denote validation data, `n` is the maximum number of trees grown and `cat` is a list of indices of categorical variables. Training and validation data are represented as matrices, with one row corresponding to one data sample. The role of parameters `p` differs for each method and will be explained separately in subsequent sections.

Method `Learn(Xt, Yt, cat, n, p)` trains a fixed number of `n` trees using training data `Xt`, `Yt` with training parameters `p`. Learning with early stopping with a preset upped bound on the total number of trees is handled by method `LearnByEarlyStoppingOnOutOfBagData(Xt, Yt, cat, n, iter, p)`, where `iter` is the number of iterations without an improvement in the out-of-bag error estimate for which the algorithm terminates.
Analogously, method `LearnByEarlyStoppingOnValidationSet(Xt, Yt, Xv, Yv, cat, n, iter, p)` handles learning with early stopping for those cases where we have a separate validation set `Xv`, `Yv` to estimate generalization error.

Method `Evaluate(X)` is responsible for evaluation of tree ensemble responses. Resubstitution, validation, out-of-bag and test set errors are calculated by methods `ResubstitutionError()`, `ValidationSetError()`, `OutOfBagError()` and `TestSetError(Xt,Yt)`. Instead of returning a single number representing the total mean squared error, these error functions return an instance of the `TreeEnsembleError` class, which aside from mean squared error for the whole ensemble also records mean squared errors for individual trees and for all subensembles. As in learning, data paremeters `X`, `Xt`, `Yt` are expected in form of matrices.

The `CalculateProximityMatrix(X)` method calculates proximity matrix for a set of samples `X`.

4.1.1 Bagging

Bagging of regression trees is implemented in the `TreeBagging` class. Thanks to the existing regression tree implementation in the `classregtree` class, the implementation is quite straightforward; a sequence of `classregtree` trees is trained with parameters `prune='off'` to disable storing of the pruning information and `nvartosample='all'` to avoid splitting on a random subset of the variables.

Two types of sampling are implemented for picking training samples from the original training data:

- M samples are picked from the training set of size N by sampling with repetition. This is the standard approach along with $M = N$.
- $M \leq N$ samples are picked from the training set of size N by sampling without repetition. This approach is implemented mainly for experimentation purposes.

Parameters for `TreeBagging` are encapsulated in the `TreeBaggingParameters` class, which has the following properties:

- `SamplingMethod` - sampling method, either `'repetition'` for sampling with repetition or `'norepetition'` for sampling without repetition. Default value is `'repetition'`.

- `SampleRelativeSize` - relative size of the training set for one particular tree with respect to the size of the original training set, a positive real number. Default value is 1.
- `MinLeaf` - `minleaf` parameter for regression trees grown using `classregtree`. Default value is 1.
- `MinParent` - `minparent` parameter for regression trees grown using `classregtree`. Default value is 2.

4.1.2 Random Forests

Random forests are implemented in the `RandomForest` class similarly to bagging; only this time when growing regression trees via `classregtree`, `nvartosample` parameter must be set appropriately to enable splitting based on random subsample of all variables.

Two types of sampling are implemented for picking training samples from the original training data:

- M samples are picked from the training set of size N by sampling with repetition. This is the standard approach along with $M = N$.
- $M \leq N$ samples are picked from the training set of size N by sampling without repetition. This approach is implemented mainly for experimentation purposes.

Parameters for `RandomForest` are encapsulated in the `RandomForestParameters` class, which has the following properties:

- `SamplingMethod` - sampling method, either `'repetition'` for sampling with repetition or `'norepetition'` for sampling without repetition. Default value is `'repetition'`.
- `SampleRelativeSize` - ratio between the size of the training set for one particular tree and the size of the original training set, a positive real number. Default value is 1.
- `VariableSampleRelativeSize` - ratio between the number of variables considered when picking the best split of a tree node and the total number of variables in the training data. Real number from the interval $(0, 1]$, default value is $\frac{1}{3}$.
- `MinLeaf` - `minleaf` parameter for regression trees grown using `classregtree`. Default value is 1.
- `MinParent` - `minparent` parameter for regression trees grown using `classregtree`. Default value is 2.

4.1.3 AdaBoost R2

AdaBoost R2 for regression trees is implemented in the `TreeAdaBoostR2` class. As in bagging and random forests, the implementation relies on `classregtree` trees, this time with parameters `prune='off'` to disable storing of the pruning information and `nvartosample='all'` to avoid splitting on a random subset of the variables. An option would be to prune the trees to a small fixed number of leaves just like in stochastic gradient boosting, but this approach was almost immediately turned down for its poor performance.

Although Drucker in his publication ([7]) does not explicitly stress this fact, weights on training samples need to be normalized in each iteration. Not doing so results in numeric instability and crashes of the algorithm.

AdaBoost R2 does not work with out-of-bag data the way bagging and random forests do and, as a consequence, related learning and error computation methods are not supported.

Parameters for `TreeAdaBoostR2` are encapsulated in the `TreeAdaBoostR2Parameters` class, which offers the following properties:

- `MinLeaf` - `minleaf` parameter for regression trees grown using `classregtree`. Default value is 1.
- `MinParent` - `minparent` parameter for regression trees grown using `classregtree`. Default value is 2.

4.1.4 Stochastic Gradient Boosting

Stochastic gradient boosting based on regression trees is implemented in the `TreeStochasticGradientBoosting` class. Again, `classregtree` class can be used to grow regression trees, although this time the approach has one major drawback - regression tree implementation in the `classregtree` class does not support growing trees with a fixed number of leaves. The greedy splitting process in `classregtree` proceeds in top-down, left-to-right manner and nodes for potential splitting are processed using a first-in first-out node buffer. In order to grow a tree with a fixed number of leaves, a different approach is necessary - at one time, we need to search for the split producing the biggest gain among all nodes and possible splits.

As it would be hard to modify the `classregtree` implementation, a different approach was adopted - first, a full unpruned tree is grown using `classregtree`, then the recursive partitioning process is simulated starting from the root. For each node in the tree, `classregtree` has already chosen the optimal split, so in each step of our simulated splitting proceess, we only have to find a leaf and an split giving the biggest decrease in the total error across all possible splits of all current leaf nodes. This way we can identify subtree with exactly N leaves in N steps. Needless to say, the mentioned approach is computationally not optimal and its only purpose is overcoming a weakness in `classregtree` regression tree implementation. Pruning to a fixed number of leaves is implemented in the `TreeHelper.PruneTreeToFixedNumberOfLeaves` function.

Two types of sampling are implemented for picking training samples from the original training data:

- $M \leq N$ samples are picked from the training set of size N by sampling without repetition. This is the standard approach for stochastic gradient boosting.

- M samples are picked from the training set of size N by sampling with repetition. This approach is implemented purely for experimentation purposes.

Stochastic gradient boosting cannot work with out-of-bag data the way bagging and random forests do and, as a consequence, related learning and error computation methods are not supported.

Parameters for `TreeStochasticGradientBoosting` are encapsulated in the `TreeStochasticGradientBoostingParameters` class, which offers the following properties:

- `SamplingMethod` - sampling method, either `'repetition'` for sampling with repetition or `'norepetition'` for sampling without repetition. Default value is `'norepetition'`.

- `SampleRelativeSize` - relative size of the training set for one particular tree with respect to the size of the original training set, a positive real number. Default value is 0.5.

- `NumberOfLeaves` - number of leaves in trained regression trees, a positive integer ≥ 2. Default value is 6.

- `Shrinkage` - value of the shrinkage parameter υ which controls the learning rate of the algorithm, a positive real number. Default value is 0.01.

4.2 Genetic Algorithm

Parts of the genetic algorithm (selection, crossover and mutation operators, fitness scaling, ...) are implemented in separate classes using inheritance to exactly match the structure of the genetic algorithm as proposed in Chapter 3. The list of all relevant classes in presented in the Table 4.2, along with classes containing polytope routines.

The genetic algorithm itself is implemented in the `GA` class and its `Run` method. Property `Parameters` of the `GA` class, which is an instance of the `GAParameters` class, represents parameters of the genetic algorithm. It contains the following properties:

- `MaxGenerations` - maximum number of iterations of the genetic algorithm, a positive integer. Default value is 1000.

- `MaxEvaluations` - maximum number of objective function evaluations, a positive integer. Default value is 5000.

- `MinDatabaseSizeForSurrogateModel` - minimum number of objective function samples required to construct the surrogate model, a positive integer. Default value is 200.

Group	Base class	Derived classes
Crossover operators	`CrossoverOperator`	`CrossoverOperatorWithMutation` `CrossoverOperatorArithmetic`

Mutation operators	`MutationOperator`	`MutationOperatorContinuousStep`
		`MutationOperatorContinuousBall`
		`MutationOperatorContinuousCones`
		`MutationOperatorDiscreteContinuous`

Selection operators	`SelectionOperator`	`SelectionOperatorTournament`
		`SelectionOperatorProportionate`
		`SelectionOperatorStochasticUniform`

Fitness scaling operators	`FitnessScaling`	`FitnessScalingSimple`
		`FitnessScalingLinear`
		`FitnessScalingRank`

Individual based control	`IndividualBased-Control`	`IndividualBasedControlBest`
		`IndividualBasedControlRandom`
		`IndividualBasedControlRandomWeighted`
		`IndividualBasedControlClustering`

Random population generators	`PopulationGenerator`	`PopulationGeneratorRandom`
Surrogate models	`SurrogateModel`	`SurrogateModelSingle` `SurrogateModelTournament`
Database	`Database`	-
Polytope computations	-	`BoundedPolytope` `BoundedPolytopeProjection`

- `SurrogateModel` - surrogate model, an instance of a class deriving from the `SurrogateModel` class.
- `ObjectiveFunction` - objective function formula as a lambda expression in *MATLAB* syntax `@(x)f(x)` (for example, `@(x)x^2`).
- `EquivalenceClasses` - array of polytope equivalence classes, obtained by parsing a formal language used to define optimization problems in catalysis.
- `FitnessScaling` - fitness scaling operator, an instance of a class deriving from the `ScalingOperator` class or, if there is no fitness scaling, an empty array.
- `SelectionOperator` - selection operator, an instance of a class deriving from `SelectionOperator` class.
- `GeneticOperators` - cell array of genetic operators, all instances must derive from either `CrossoverOperator` or `MutationOperator` classes.
- `GeneticOperatorsWeights` - array of fractions of the offspring created by genetic operators. Must have the same length as `GeneticOperators`, each

value being from the interval $(0, 1]$. Sum of all elements must be equal to 1.

- `Database` - instance of the `Database` class, contains samples evaluated by the objective function.
- `PopulationSize` - population size for the genetic algorithm, a positive integer. Default value is 100.
- `InitialPopulation` - initial population in the form of a matrix, each solution represented by one row. Empty array in the case there is no initial population.
- `InitialFitness` - initial fitness (only applicable if `InitialPopulation` is also specified), a vector of fitness values or an empty array.
- `PopulationGenerator` - generator of the initial population, an instance of the `PopulationGenerator` class or an empty array.
- `EvolutionControlType` - chosen evolution control type as defined by the `EvolutionControlTypes` enumeration, taking values of `None`, `IndividualBased` and `GenerationBased`. Default value is `None`.
- `GenerationBasedControlCycleGenerations` - length of the generation-based control cycle, a positive integer. Default value is 3.
- `GenerationBasedControlModelGenerations` - number of generations evaluated using the model within one cycle of the generation-based control, a positive integer smaller than `GenerationBasedControlCycleGenerations`. Default value is 2.
- `IndividualBasedControl` - cell array of individual-based control operators, each instance must derive from the `IndividualBasedControl` class.
- `IndividualBasedControlWeights` - array of fractions of offspring evaluated using the objective function by operators specified in `IndividualBasedControl`. Must have the same length as `IndividualBasedControl`, each value being from the interval $(0, 1]$. Sum of all elements must be equal to 1.
- `IndividualBasedControlMultiplier` - relative size of the offspring population compared to regular population in the case of individual-based evolution control, a real number greater than 1. Default value is 10.
- `EliteCount` - number of individuals picked by elitism into the next generation, a positive integer. Default value is 2.

In one run of the genetic algorithm, properties `MinFunctionValues` and `MeanFunctionValues` of the `GA` class are filled with minimum and mean function values observed in each generation of the optimization algorithm. Method `[x mi me]=FunctionValuesByEvaluations()` on the `Database` class can be used to calculate minimum and mean function values at each moment when new data samples were evaluated by the objective function. In this case, `x` is a vector

of database sizes at each moment new data samples were added and `mi` and `me` are vectors of the same length as `x` containing minimum and mean values of the objective function.

5. Experiments

5.1 Tree Ensembles

The prototype implementation of tree ensembles has been tested on several datasets and artificial benchmark functions. This section reviews the testing methodology and the obtained results, which can be used to compare performance of tree ensembles to other regression methods.

Even more importantly, the goal of the testing phase was to observe the behavior of different learning methods for all implemented types of tree ensembles. Performance of learning with early stopping rules needs to be assessed for many reasons - to test soundness of the early stopping rules, computational savings they can offer and to observe how much the implemented models tend to overfit on the training data.

In optimization of empirical objective functions, where the number of available function samples is usually quite limited, splitting data between training and validation sets may have a negative impact on the model performance, which stresses the need for reliable error estimation based on out-of-bag data.

Effectiveness of the implemented ensemble methods (measured by the number of trees and tree nodes required to accurately fit the model on the training data) is also quite relevant for practical purposes.

5.1.1 Testing Methodology

Results are presented from benchmarking of all the methods described in Chapter 2.1, ie. regression trees, bagging, random forests, stochastic gradient boosting and AdaBoost R2. Particularly, the following models have been tested:

- Single regression tree pruned using crossvalidation.
- Single regression tree pruned using a separate validation set.
- Bagged tree ensemble trained using three different training methods.
- Random forest trained using three different training methods.
- Stochastic gradient boosting ensemble trained using three different training methods.
- AdaBoost R2 ensemble trained using three different training methods.

For tree ensembles, the tested training methods have been:

- Training the ensemble to a fixed size of 500 trees.
- Training the ensemble to no more than 500 trees with an early stopping rule using a separate validation set.
- Training the ensemble to no more than 500 trees with an early stopping rule using out-of-bag data (only applicable to bagging and random forests).

Due to heavy CPU time requirements even on modern multi-core processors, the models could only be reasonably tested with default parameter values specified in Section 4.1.

The presented figures show mean squared errors, ensemble sizes and ensemble node counts averaged over 10 runs of the benchmark procedure, along with standard deviations of the obtained results.

5.1.2 Performance on Benchmark Functions

Three benchmark functions proposed by Friedman in his work on Multivariate Adaptive Regression Splines [16] were chosen to test prediction capabilities of the models. Definition of these functions is presented in Table 5.1. All three function incorporate random noise from normal distribution with zero mean and adjustable standard deviation. For the purpose of the testing, $\sigma_1 = 1$, $\sigma_2 = 125$ and $\sigma_3 = 0.1$ were used.

Function	Definition
Friedman #1	$f_1(x_1, \ldots, x_{10}) = 10sin(\pi x_1 x_2) + 20(x_3 - 0.5)^2 + 10x_4 + 5x_5 + \mathcal{N}(0, \sigma_1^2)$ $x_i \in [0, 1], i = 1, \ldots, 10$
Friedman #2	$f_2(x_1, \ldots, x_4) = \sqrt{x_1^2 + (x_2 x_3 - \frac{1}{x_2 x_4})^2} + \mathcal{N}(0, \sigma_2^2)$ $x_1 \in [0, 100], x_2 \in [40\pi, 560\pi], x_3 \in [0, 1], x_4 \in [1, 11]$
Friedman #3	$f_3(x_1, \ldots, x_4) = tan^{-1}(x_1^{-1}(x_2 x_3 - \frac{1}{x_2 x_4})) + \mathcal{N}(0, \sigma_3^2)$ $x_1 \in [0, 100], x_2 \in [40\pi, 560\pi], x_3 \in [0, 1], x_4 \in [1, 11]$

Table 5.1: Definition of functions Friedman #1, #2 and #3.

In each of the 10 runs for one particular benchmark function, this function was sampled in 1000 random points. To benchmark training methods that require a separate validation set, these 1000 samples were split in 2:1:1 ratio between the training, validation and test sets. For the testing of the remaining training methods, 1000 samples were split in 1:1 ratio between the training and the test set.

When measuring mean squared error on the test set, the error was calculated by comparing model responses to responses from the generated test set, excluding the effect of the random noise. Such approach appears to be more reasonable than testing on noisy data and should reliably estimate the generalization error.

Benchmark results (mean square errors, ensemble sizes and ensemble node counts) are contained in Table 5.2, Table 5.3 and Table 5.4.

Obviously, introducing tree ensembles in place of isolated regression trees brings a huge performance improvement. In terms of mean-squared error, stochastic gradient boosting consistently outperforms all three remaining ensemble methods. Bagging and random forests come second for two of three functions, random forests performing slightly worse than bagging in almost all cases.

AdaBoost R2 gives uneven results, performing better than bagging and random forests on Friedman #1 and much worse on the remaining two functions. In a quick experiment, 10 AdaBoost R2 ensembles of 500 trees trained with modified parameter values of `MinLeaf=5` and `MinParent=10` performed much better on Friedman functions #2 and #3 with mean squared error of $37.94 \pm 4.28 \times 10^2$ and $7.79 \pm 2.59 \times 10^{-3}$ respectively (compared to values of $65.52 \pm 6.66 \times 10^2$ and

$8.98 \pm 2.56 \times 10^{-3}$ from the Table 5.3). Even then however, the descrease in error was not sufficient to beat bagged ensembles.

Both early stopping rules (using out-of-bag data and a separate validation set) performed well and strongly reduced the size of the resulting ensembles, usually a lot below 200 trees. For stochastic gradient boosting specifically, these rules provide little or no reduction in number of trees due to the fact that stochastic gradient boosting converges slowly and does not tend to overfit on the training data so much, thus reaching the maximum ensemble size of 500 trees anyway in most cases.

However, stochastic gradient boosting compensates for the large number of trees by small number of nodes in each tree and when measuring the ensemble complexity in terms of number of contained nodes, stochastic gradient boosting wins by a large margin. Compared to stochastic gradient boosting, bagging and random forests do at least twenty times worse and AdaBoost R2 stands somewhere in between.

Overall, stochastic gradient boosting seems to achieve the best mean squared error among the tested ensemble methods and is also a clear winner in terms of complexity of the resulting model measured by the number of tree nodes. Bagging and random forests are good for creating ensembles having small numbers of trees as, compared to stochastic gradient boosting, these methods converge faster. It is hard to draw any conclusions about AdaBoost R2 because of its unconvincing performance in this test.

Tree count		B	RF	SGB	ABR2
Friedman #1	(VS)	75 ± 48	67 ± 37	500 ± 1	72 ± 41
	(OOB)	135 ± 70	123 ± 45	-	-
Friedman #2	(VS)	53 ± 32	58 ± 36	439 ± 58	32 ± 15
	(OOB)	126 ± 55	103 ± 38	-	-
Friedman #3	(VS)	29 ± 19	44 ± 44	494 ± 19	29 ± 15
	(OOB)	72 ± 40	118 ± 63	-	-

Table 5.2: Benchmark of tree ensembles on Friedman functions - ensemble sizes. B - bagging, RF - random forest, SGB - stochastic gradient boosting, ABR2 - AdaBoostR2. (VS) - ensemble trained with early stopping rule using a validation set, (OOB) - ensemble trained with early stopping rule using out-of-bag data.

MSE	RT1	RT2		B	RF	SGB	ABR2
Friedman #1	8,77 ± 0,77	8,60 ± 1,05	(500)	3,59 ± 0,30	3,60 ± 0,22	1,99 ± 0,16	3,00 ± 0,18
			(VS)	3,83 ± 0,43	3,89 ± 0,43	2,07 ± 0,29	3,39 ± 0,52
			(OOB)	3,63 ± 0,32	3,67 ± 0,23		
Friedman #2	97,99 ± 15,83	95,19 ± 19,09	(500)	34,57 ± 4,16	39,03 ± 7,54	19,11 ± 3,98	65,52 ± 6,66
			(VS)	35,73 ± 5,62	43,11 ± 7,90	17,46 ± 3,11	66,14 ± 11,00
			(OOB)	35,80 ± 4,90	41,62 ± 8,40		
Friedman #3	22,15 ± 4,37	24,33 ± 8,29	(500)	6,81 ± 1,97	7,30 ± 1,45	4,39 ± 1,04	8,98 ± 2,56
			(VS)	8,37 ± 1,88	9,04 ± 2,15	4,83 ± 1,14	10,32 ± 3,13
			(OOB)	7,20 ± 2,10	7,60 ± 1,51		

Table 5.3: Benchmark of tree ensembles on Friedman functions - mean squared error. RT1 - regression tree pruned using cross-validation, RT2 - regression tree pruned using a validation set, B - bagging, RF - random forest, SGB - stochastic gradient boosting, ABR2 - AdaBoost R2. (500) - ensemble trained to 500 trees, (VS) - ensemble trained with early stopping rule using a validation set, (OOB) - ensemble trained with early stopping rule using out-of-bag data.

MSE	RT1	RT2		B	RF	SGB	ABR2
Friedman #1	36 ± 14	36 ± 15	(500)	312 826 ± 483	313 755 ± 421	5 500 ± 0	200 949 ± 3 725
			(VS)	46 669 ± 29 685	42 092 ± 22 887	5 497 ± 7	31 048 ± 17 014
			(OOB)	84 236 ± 43 482	77 389 ± 28 127		
Friedman #2	34 ± 5	35 ± 8	(500)	312 510 ± 737	313 850 ± 468	5 500 ± 0	193 710 ± 2 788
			(VS)	33 099 ± 20 276	36 319 ± 22 382	4 828 ± 638	14 236 ± 5 782
			(OOB)	78 480 ± 34 620	64 667 ± 23 851		
Friedman #3	33 ± 7	34 ± 7	(500)	312 880 ± 470	313 901 ± 318	5 500 ± 0	148 030 ± 6 912
			(VS)	18 369 ± 11 786	27 676 ± 27 479	5 432 ± 212	11 349 ± 5 141
			(OOB)	44 858 ± 24 872	73 865 ± 39 629		

Table 5.4: Benchmark of tree ensembles on Friedman functions - number of nodes. RT1 - regression tree pruned by crossvalidation, RT2 - regression tree pruned using a validation set, B - bagging, RF - random forest, SGB - stochastic gradient boosting, ABR2 - AdaBoostR2. (500) - ensemble trained to 500 trees, (VS) - ensemble trained with early stopping rule using a validation set, (OOB) - ensemble trained with early stopping rule using out-of-bag data.

5.1.3 Performance on Benchmark Datasets

In this section, performance of regression trees and tree ensembles was measured on seven often cited datasets, which were obtained from UCI[1], mltest[2] and Delve[3] repositories. Particularly, these were:

- `Abalone` (4177 samples, 1 categorical and 7 numeric variables) Data set representative for the problem of abalone age prediction based on physical measurements such as weight, height, length or sex.
- `AutoMpg` (398 samples from which 6 are removed due to missing attribute values, 3 categorical and 5 numeric variables) Data set concerning prediction of fuel consumption of cars in miles per gallon in relation to characteristics such as horse-power, weight, acceleration etc.
- `Bank32nh` (8192 samples, 32 numeric variables) Data set from a family of data sets synthetically generated from a simulation of how bank customers choose their banks depending on inputs such as distance to the bank, type of the task or their level of patience. 'nh' suffix indicates nonlinear dynamics with high amount of noise.
- `BodyFat` (252 samples, 13 numeric variables) Data set addressing the problem of estimating percentage of body fat determined by underwater weighting and various body circumference measurements for 252 men. Percentage of the body fat is used as the target variable and the density measured by underwater weighting is excluded from the data set due to redundancy (percentage of the body fat and the density measured by underwater weighting are related through a simple formula).
- `BostonHousing` (506 samples, 1 categorical variable, 12 numeric variables) Data set describing value of owner-occupied houses in the suburbs of Boston depending on town crime rates, accessibility of highways, property taxes etc.
- `ConcreteStrength` (1030 samples, 8 numeric variables) Data set for the problem of concrete compressive strength prediction based on used ingredients and age.
- `Pumadyn32nm` (8192 samples, 32 numeric variables) Data set from a family of data sets synthetically generated in a simulation of dynamics of a Puma 560 robot arm on inputs such as angular positions, velocities and torques of the robot arm. 'nm' suffix indicates nonlinear dynamics with medium amount of noise.

In each of the 10 runs for one particular data set, the data were permuted first. To benchmark training methods that require a separate validation set, data samples were split in 2:1:1 ratio between the training, validation and test sets. For the testing of the remaining training methods, data samples were split in 1:1 ratio between the training and the test set.

[1] hosted at `http://www.ics.uci.edu/~mlearn`

[2] hosted at `http://www.mldata.org/`

[3] hosted at `http://www.cs.toronto.edu/~delve/data/datasets.html`

Benchmark results (mean square errors, ensemble sizes and ensemble node counts) are contained in Table 5.5, Table 5.6 and Table 5.7.

Compared to results of benchmarking on artificial test functions, the interpretation of results obtained on real-world data sets seems to be more difficult. In terms of mean squared error, stochastic gradient boosting outperforms other ensemble methods on `AutoMpg`, `BodyFat`, `Bank32nh` and, ignoring the high variance of the results, `BostonHousing` data sets. On `Abalone` data set, it seems to be on-par with bagging and random forests; on `ConcreteStrength`, it loses to these two ensemble methods. Considering the fact that for `Abalone` and `ConcreteStrength` data sets, stochastic gradient boosting almost always trains the maximum number of 500 trees even when using early stopping rules, there is a chance that the prediction ability for these data sets may be considerably improved by simply adding more trees to the ensembles. As with artificial test functions, performance of AdaBoost R2 is rather inconsistent - for some data sets, it is comparable to that of bagging and random forests, for some it is considerably worse and on `Pumadyn32nm` data set, it beats all other methods.

Results on `Pumadyn32nm` data set require special attention, as the mean squared error of random forests and stochastic gradient boosting is actually worse that that obtained using a single regression tree. Few simple experiments were run to see if it is possible to obtain better results with different parameter choices.

Keeping the default parameter values, stochastic gradient boosting ensemble was trained to the size of 2000 trees to check converge speed, showing that it is indeed very slow, ensemble of 1000 trees produced mean squared error of 8.12×10^5 and for 1500 and 2000 trees, values of 7.72×10^5 and 7.52×10^5 were obtained. Increased value of the shrinkage parameter (0.005 and 0.01) with only 500 trees performed better than the large ensemble with mean squared errors of 7.37×10^5 and 7.36×10^5. Setting number of leaves of the constructed trees to 8 and 10 yielded worse results than those obtained with different ensemble sizes or shrinkage. Although these are results of a single experiment instead of ten, they should be pretty reliable considering the low variance of mean squared error observed with this data set. Altogether, `Pymadyn32nm` seems to be a difficult problem for stochastic gradient boosting.

The situation with random forests is easier: the issue seem to be the choice of the number of variables considered when splitting tree nodes. With the default parameter values, which result in choice of 11 random variables for each split, the algorithm is unable to cope with the data, most likely explanation being that the output is heavily dependent only on a small number of predictors. The mean squared error improves steadily with increasing values of `VariableSampleRelativeSize` and the choice of `VariableSampleRelativeSize` equal approximately to 0.6 results in mean squared error of $6.35 \pm 0.16 \times 10^{-5}$, which is comparable to bagging (this result was obtained by training 10 random forests to a fixed size of 500 trees using the forementioned parameters).

Yet again, early stopping rules produce models of high quality. Compared to the results from artificial test functions, the difference in obtained ensemble sizes is even more accentuated between stochastic gradient boosting and other ensemble methods. Early stopping rules employed with bagging and random forests often produce models of size less than 100 trees, whereas stochastic gradient boosting requires hundreds of iterations to converge. On the other hand,

when measuring complexity by the obtained numbers of tree nodes, stochastic gradient boosting totally outperforms all the remaining methods.

Testing on data sets does not have a clear winner and shows that even the simplest ensemble model, bagged ensembles, can produce high quality results.

Tree count		B	RF	SGB	ABR2
Abalone	(VS)	81 ± 34	81 ± 51	499 ± 3	39 ± 26
	(OOB)	113 ± 21	150 ± 42	-	-
AutoMpg	(VS)	36 ± 34	44 ± 51	399 ± 3	30 ± 26
	(OOB)	84 ± 21	74 ± 42	-	-
Bank32nh	(VS)	100 ± 36	101 ± 31	500 ± 1	66 ± 32
	(OOB)	160 ± 39	174 ± 72	-	-
BodyFat	(VS)	43 ± 26	36 ± 26	289 ± 54	36 ± 25
	(OOB)	59 ± 36	80 ± 48	-	-
Housing	(VS)	22 ± 12	23 ± 13	447 ± 70	34 ± 23
	(OOB)	67 ± 39	79 ± 34	-	-
Concrete	(VS)	46 ±31	55 ±32	500 ±0	49 ±32
	(OOB)	87 ±40	97 ±31	-	-
Pumadyn32nm	(VS)	140 ± 74	66 ± 44	500 ± 0	134 ± 49
	(OOB)	189 ± 45	85 ± 33	-	-

Table 5.5: Benchmark of tree ensembles on often cited data sets - ensemble sizes. B - bagging, RF - random forest, SGB - stochastic gradient boosting, ABR2 - AdaBoost R2. (VS) - ensemble trained with early stopping rule using a validation set, (OOB) - ensemble trained with early stopping rule using out-of-bag data.

MSE	RT1	RT2		B	RF	SGB	ABR2
Abalone	6,01 ± 0,28	5,84 ± 0,32	(500)	4,90 ± 0,21	4,74 ± 0,18	4,85 ± 0,18	5,42 ± 0,29
			(VS)	4,85 ± 0,38	4,76 ± 0,35	4,82 ± 0,27	5,38 ± 0,36
			(OOB)	4,92 ± 0,22	4,77 ± 0,16		
AutoMpg	15,13 ± 2,46	14,42 ± 2,50	(500)	8,94 ± 1,14	9,12 ± 1,29	8,20 ± 1,10	9,61 ± 1,57
			(VS)	9,73 ± 1,84	9,20 ± 1,87	8,60 ± 2,01	10,45 ± 1,98
			(OOB)	9,05 ± 1,10	9,29 ± 1,42		
BodyFat	31,76 ± 1,62	29,18 ± 6,34	(500)	23,13 ± 1,28	23,91 ± 1,12	21,36 ± 0,72	23,70 ± 1,63
			(VS)	26,25 ± 4,05	26,92 ± 4,29	23,84 ± 3,42	27,70 ± 3,25
			(OOB)	23,39 ± 1,43	24,31 ± 1,23		
Housing	29,48 ± 6,96	22,56 ± 5,69	(500)	16,54 ± 6,76	15,93 ± 5,60	14,45 ± 5,37	15,99 ± 6,18
			(VS)	12,30 ± 3,32	12,43 ± 3,31	11,42 ± 3,60	12,99 ± 4,92
			(OOB)	16,72 ± 6,67	16,09 ± 5,28		
Concrete	68,37 ± 9,00	72,18 ± 14,11	(500)	33,12 ± 4,60	33,34 ± 3,96	34,69 ± 3,61	36,66 ± 4,09
			(VS)	33,85 ± 3,33	34,91 ± 4,01	35,72 ± 2,51	35,57 ± 2,41
			(OOB)	34,00 ± 4,64	33,63 ± 3,91		
Bank32nh	9,27 ± 0,27	9,26 ± 0,34	(500)	7,36 ± 0,10	7,36 ± 0,14	7,15 ± 0,14	7,67 ± 0,21
			(VS)	7,40 ± 0,30	7,42 ± 0,30	7,10 ± 0,28	7,90 ± 0,30
			(OOB)	7,39 ± 0,11	7,40 ± 0,13		
Pumadyn32nm	8,15 ± 0,20	7,97 ± 0,18	(500)	6,29 ± 0,12	12,08 ± 0,67	9,31 ± 0,20	6,23 ± 0,13
			(VS)	6,28 ± 0,16	11,66 ± 0,68	9,49 ± 0,38	6,34 ± 0,19
			(OOB)	6,31 ± 0,11	11,82 ± 0,62		

Table 5.6: Benchmark of tree ensembles on often cited data sets - mean squared error. RT1 - regression tree pruned using cross-validation, RT2 - regression tree pruned using a validation set, B - bagging, RF - random forest, SGB - stochastic gradient boosting, ABR2 - AdaBoost R2. (500) - ensemble trained to 500 trees, (VS) - ensemble trained with early stopping rule using a validation set, (OOB) - ensemble trained with early stopping rule using out-of-bag data.

MSE	RT1	RT2		B	RF	SGB	ABR2
Abalone	19 ± 3	18 ± 4	(500)	799 150 ± 6 299	917 682 ± 5 414	5 500 ± 0	489 875 ± 8 549
			(VS)	129 330 ± 53 960	148 986 ± 94 445	5 484 ± 29	44 163 ± 25 611
			(OOB)	180 427 ± 34 534	274 597 ± 77 004		
AutoMpg	12 ± 3	13 ± 3	(500)	89 319 ± 2 038	98 621 ± 1 363	5 500 ± 0	46 423 ± 3 753
			(VS)	6 406 ± 4 360	8 762 ± 7 705	4 393 ± 657	3 218 ± 2 441
			(OOB)	15 000 ± 7 930	14 507 ± 7 150		
BodyFat	6 ± 2	8 ± 2	(500)	77 309 ± 541	77 906 ± 454	5 500 ± 0	47 424 ± 1 533
			(VS)	6 654 ± 3 945	5 628 ± 4 013	3 180 ± 598	3 791 ± 2 475
			(OOB)	9 123 ± 5 475	12 489 ± 7 605		
Housing	11 ± 4	16 ± 11	(500)	151 445 ± 1 161	153 296 ± 1 029	5 500 ± 0	60 685 ± 9 069
			(VS)	6 559 ± 3 604	7 075 ± 4 000	4 918 ± 774	4 659 ± 2 427
			(OOB)	20 398 ± 11 985	24 320 ± 10 479		
Concrete	76 ± 21	66 ± 24	(500)	307 765 ± 2 090	284 296 ± 2 248	5 500 ± 0	146 400 ± 49 165
			(VS)	28 519 ± 19 013	30 929 ± 18 225	5 500 ± 0	15 495 ± 11 590
			(OOB)	53 427 ± 24 649	54 838 ± 17 760		
Bank32nh	17 ± 3	17 ± 3	(500)	2 115 998 ± 11 501	2 162 887 ± 11 310	5 500 ± 0	1 258 808 ± 14 912
			(VS)	421 779 ± 151 997	436 909 ± 132 645	5 498 ± 7	184 870 ± 81 435
			(OOB)	677 559 ± 167 578	752 780 ± 315 606		
Pumadyn32nm	141 ± 9	151 ± 14	(500)	2 514 633 ± 2 935	2 546 995 ± 1 628	5 500 ± 0	1 773 607 ± 19 272
			(VS)	705 040 ± 374 973	337 128 ± 223 890	5 499 ± 3	490 803 ± 174 004
			(OOB)	947 964 ± 224 201	434 386 ± 168 008		

Table 5.7: Benchmark of tree ensembles on often cited data sets - number of nodes. RT1 - regression tree pruned using cross-validation, RT2 - regression tree pruned using a validation set, B - bagging, RF - random forest, SGB - stochastic gradient boosting, ABR2 - AdaBoost R2. (500) - ensemble trained to 500 trees, (VS) - ensemble trained with early stopping rule using a validation set, (OOB) - ensemble trained with early stopping rule using out-of-bag data.

5.1.4 Performance on the HCN Dataset

Finally, the results of testing of the proposed surrogate model on a data set from a catalysis experiment ([30]) are presented, in which different combinations of active elements and support compounds were tried to maximize yields of hydrocyanic acid produced in a reaction of methane and ammonia.

The problem consists of 11 continous variables $x_1^c, \ldots, x_{11}^c$ corresponding to proportions of 11 different metals (Y, La, Zr, Mo, Re, Ir, Ni, Pt, Zn, Ag and Au), each x_i^c being from the $[0, 1]$ interval and $\sum_{i=1}^{11} x_i^c = 1$. The number of active metal elements (those elements whose proportions are greater than zero) is given by discrete numeric variable x_c^d. Support, represented by discrete variable x_s^d, is formed by 15 different chemical compounds (Si_3N_4,Sm_2O_3,...). The target variable y represents the percentage of hydrocyanic acid yield in proportion to the amount of reacting ammonia.

The data set composed of total of 701 samples $\{(x_1^c, \ldots, x_{11}^c, x_c^d, x_s^d, y)\}_{i=1}^{701}$ evaluated during seven generations of a genetic algorithm utilized in the experiment, divided between a training set for the surrogate model (containing 609 samples from the first six generations of the genetic algorithm) and a test set (containing 92 samples from the seventh generation of the genetic algorithm).

In order to train the surrogate model, bit vector $(b_1, \ldots, b_{11})$ was added to each data sample accordingly to Section 3.5 such that $b_i = 1$ if and only if continuous variable x_i^c was active.

First, six surrogate models were trained using the training data - single regression tree pruned using crossvalidation, single regression tree pruned on a validation set (obtained by splitting 609 training samples in 9 : 1 ratio between the training and validation sets) and four ensemble models (bagging, random forest, stochastic gradient boosting, AdaBoost R2), which were trained to a fixed size of 500 trees. Scatter plots showing performance of these surrogate models on the test set are displayed in Figure 5.1. Judging by these figures, using surrogate model based on a single regression tree in this experiment could not be recommended due to poor approximation capability and high variance of predictions; tree ensembles archieve a considerably better fit with less variance.

In the next step, methodology similar to that in Section 5.1.3 was used to measure performance of different surrogate models on the hydrocyanic acid data set. As this time the data were already split into training and test sets, decision had to be made about how to benchmark single regression tree pruned using a separate validation set and tree ensembles learned by early stopping on the validation set. Due to limited number of samples in the training set, the decision was made to use only 10% of training samples for validation. As before, the benchmark procedure was run 10 times.

Benchmark results (mean squared errors, ensemble sizes and ensemble node counts) are contained in Table 5.8, Table 5.9 and Table 5.10.

In terms of mean squared error, all types of tree ensembles outperform single regression trees by a big margin, stochastic gradient boosting being slightly ahead of other methods. Early stopping based on a validation set seems to produce worse models and also results in more variance in the measured error, perhaps with the exception of stochastic gradient boosting, which always creates large ensembles even with the early stopping rule. For validation sets having relative sizes of $20\%, 30\%$ and 40% compared to presented 10%, this behavior was even

more pronounced.

Overall, judging by the quality of the obtained results, there is a high chance that the proposed surrogate model based on regression tree ensembles could considerably accelerate the aforementioned experiment.

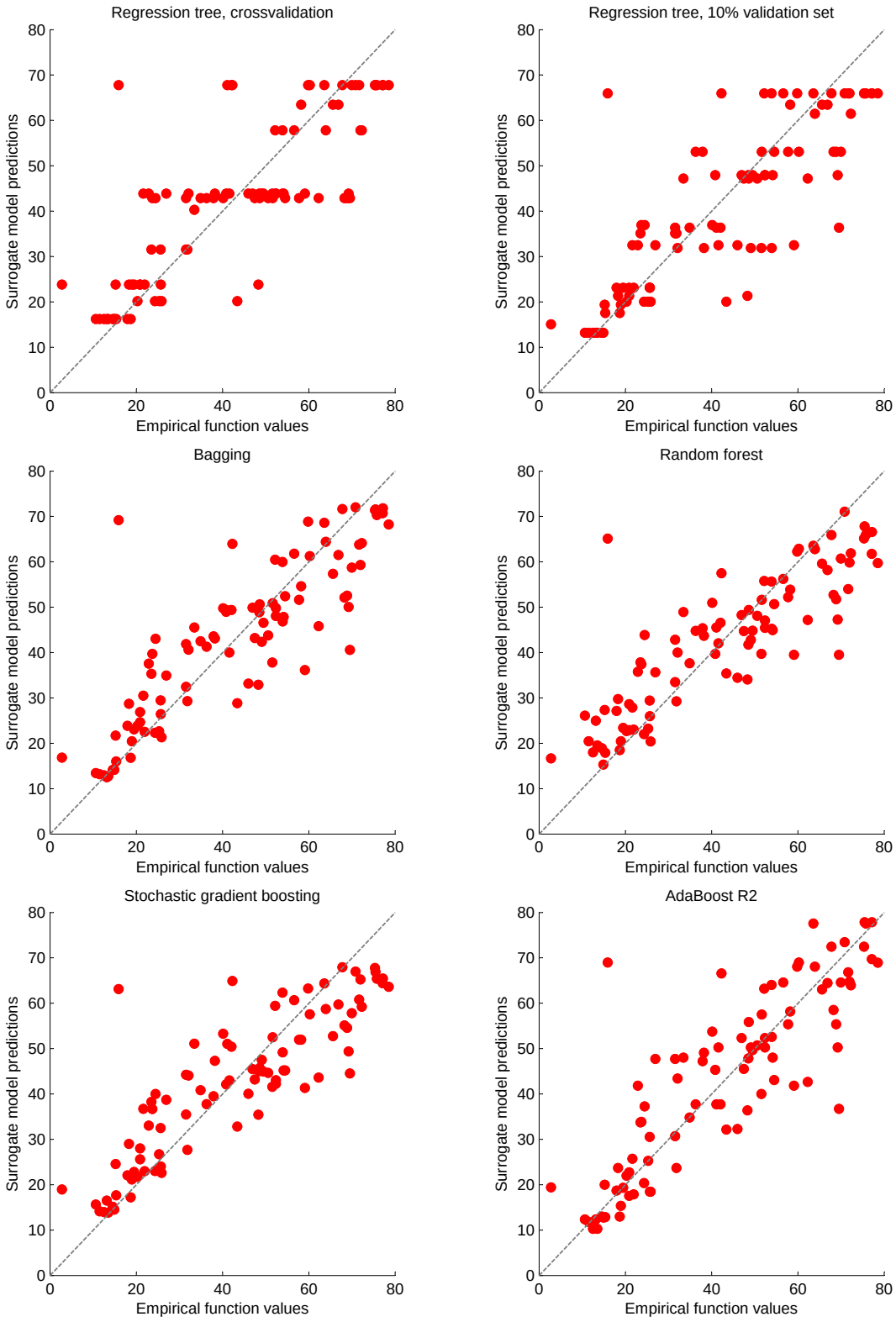


Figure 5.1: Surrogate model predictions compared to values empirically obtained during the HCN experiment. On 92 samples evaluated using the empirical objective function in the seventh generation of HCN experiment, each scatter plot shows the relation between HCN empirical function values and surrogate model predictions for one particular type of surrogate model. Two regression trees were trained using crossvalidation and a separate validation set containing 10% of all available training samples. Ensemble models were trained to a fixed size of 500 trees.

Tree count		B	RF	SGB	ABR2
HCN	(VS)	48 ± 35	36 ± 23	441 ± 68	25 ± 20
	(OOB)	93 ± 50	96 ± 68	-	-

Table 5.8: Benchmark of tree ensembles on the HCN data set - ensemble sizes. B - bagging, RF - random forest, SGB - stochastic gradient boosting, ABR2 - AdaBoost R2. (VS) - ensemble trained with early stopping rule using a validation set, (OOB) - ensemble trained with early stopping rule using out-of-bag data.

MSE	RT1	RT2		B	RF	SGB	ABR2
HCN	154,93 ± 2,50	166,50 ± 18,29	(500)	112,27 ± 1,06	119,14 ± 1,42	110,60 ± 0,95	112,14 ± 2,82
			(VS)	117,07 ± 4,62	124,90 ± 7,66	113,74 ± 3,71	126,51 ± 26,33
			(OOB)	114,28 ± 3,72	118,70 ± 4,13		

Table 5.9: Benchmark of tree ensembles on the HCN data set - mean squared error. B - bagging, RF - random forest, SGB - stochastic gradient boosting, ABR2 - AdaBoost R2. (500) - ensemble trained to 500 trees, (VS) - ensemble trained with early stopping rule using a validation set, (OOB) - ensemble trained with early stopping rule using out-of-bag data.

MSE	RT1	RT2		B	RF	SGB	ABR2
HCN	31 ± 5	16 ± 7	(500)	374 606 ± 304	352 044 ± 538	5 500 ± 0	68 729 ± 699
	-	-	(VS)	32 262 ± 23 698	23 210 ± 14 554	4 847 ± 743	6 940 ± 3 423
	-	-	(OOB)	69 695 ± 37 348	67 339 ± 47 614	-	-

Table 5.10: Benchmark of tree ensembles on the HCN data set - number of nodes. RT1 - regression tree pruned using cross-validation, RT2 - regression tree pruned using a validation set, B - bagging, RF - random forest, SGB - stochastic gradient boosting, ABR2 - AdaBoost R2. (500) - ensemble trained to 500 trees, (VS) - ensemble trained with early stopping rule using a validation set, (OOB) - ensemble trained with early stopping rule using out-of-bag data.

5.2 Genetic Algorithm

The prototype implementation of the genetic algorithm proposed in Chapter 3 which utilizes a surrogate model based on regression trees and their ensembles has been tested on two benchmark functions. The first benchmark function represents a smooth function of only continuous variables, while the second function also contains categorical variables.

The performance of the genetic algorithm with and without the use of the surrogate model was evaluated by comparing quality of solutions which were obtained under the same number of objective function evaluations. This way, we can observe not only if the surrogate model helps or not, but also if it helps throughout the whole run of the optimization algorithm or only in the starting phase, where more dramatic increases in the quality of the obtained solutions happen.

5.2.1 Testing Methodology

Each result presented in this section was obtained by running the proposed genetic algorithm 50 times with population size of 100.

Having observed the performance of tree ensemble methods presented in Section 5.1, the following regression tree methods were chosen for surrogate modelling:

- Single regression tree pruned using crossvalidation.
- Bagged tree ensemble trained with early stopping rule using out-of-bag data.
- Random forest trained with early stopping rule using out-of-bag data.
- Stochastic gradient boosting ensemble trained to a fixed size of 500 trees.
- AdaBoost R2 ensemble trained to a fixed size of 500 trees.

Tournament surrogate models were not tested.

When utilizing the surrogate model, offspring individuals in the first 2 generations were always evaluated using the objective function, so that the third generation can already use the surrogate model trained on 200 data samples.

Parameters of generation-based evolution control were set so that one of every three generations was evaluated using the objective function.

Individual-based control utilized two approaches - for the first benchmark function from catalysis, samples for reevaluation by the objective function were chosen in $1 : 1$ ratio by the best and the clustering strategy, using unweighted average distance between clusters. For the second function, samples for reevaluation by the objective function were chosen in $4 : 1$ ratio by the best and the random weighted strategy. In both cases, offspring multiplier was set to 10, so an offspring population of 1000 individuals was evaluated by the surrogate model, from which 100 individuals were reevaluated by the objective function.

The number of generations of the genetic algorithm was chosen so that always exactly 2000 samples were evaluated using the objective function (without surrogate model and with individual-based evolution control, this corresponds to

20 generations; generation-based control required few more than 50 generations to perform this number of objective function evaluations).

Stochastic universal selection has been used as the selection operator together with the default rank fitness rescaling.

Utilized genetic operators will be described for each benchmark function separately.

5.2.2 Performance on Benchmark Function From Catalysis Research

The benchmark function in this section comes from the article by Valero et. al ([37]), modelling the behavior of multi-component catalyst explored in the field of combinatorial catalysis. The optimization problem is formulated as a maximization problem of an objective function of five continuous variables

$$f(x_1, x_2, x_3, x_4, x_5) = g_1(x_1, x_2) + g_2(x_2, x_3)g_3(x_3, x_4, x_5),$$

where

$$g_1(x_1, x_2) = 0.6h(100x_1 - 35, 100x_2 - 35) + 0.75h(100x_1 - 10, 100x_2 - 10) + h(100x_1 - 35, 100x_2 - 10)$$
$$g_2(x_2, x_3) = 0.4h(100x_2 - 10, 100x_3 - 30)$$
$$g_3(x_3, x_4, x_5) = 5 + 25[1 - \sqrt{1 + (x_3 - 0.3)^2 + (x_4 - 0.15)^2 + (x_5 - 0.1)^2}]$$
$$h(u, v) = 100 - \sqrt{u^2 + v^2} + 50\frac{sin(\sqrt{u^2+v^2})}{\sqrt{u^2+v^2+0.001}},$$

subject to linear constraints $\sum_{i=1}^{5} x_i = 1$ and $x_i \geq 0, i = 1, \ldots, 5$.

As such, the function does not contain any categorical variables and can be represented by a single polytope equivalence class. It has been found empirically (by running *MATLAB* `fmincon` optimizer on the best result returned by a genetic algorithm) that the presented function attains its maximum of 547.7248 at the point $(0.3518, 0.0985, 0.2984, 0.1506, 0.1006)$. As we have formulated the genetic optimization algorithm for minimization, we will be searching for minimum of the function $-f(x_1, x_2, x_3, x_4, x_5)$.

Two genetic operators were applied to create the offspring - 60% of offspring individuals were created using the arithmetic crossover and 40% using the continuous ball mutation.

The overview of results obtained after 2000 objective function evaluations with the methodology established in Section 5.2.1 are summarized in Table 5.11 and Table 5.12.

Detailed graphs showing performance of individual-based evolution control are presented in Figures 5.2-5.5, while the performance of generation-based control is depicted in Figures 5.6-5.9. Each combination of surrogate model and evolution control type is shown in exactly two figures, both figures comparing the obtained results to the results obtained without evolution control. The difference between the two figures lies in the way the results are presented - the first figure shows the mean plus minus standard deviation of the objective function values obtained in 50 runs of the genetic algorithm. Second figure, on the other hand, shows the median, 0.159 and 0.841 quantiles of the objective function values obtained in 50 runs of the genetic algorithm.

In individual-based control, the surrogate model helps much more in a few initial generations than it does in the rest of the run of the genetic algorithm - the point at which it becomes disadvantageous to use the surrogate models ranges from approximately 600 to 1000 objective function evaluations, depending on the type of the surrogate model and whether the judgement is made based on mean/standard deviation or median/quantiles. In late generations, the results with the surrogate model of any type are usually worse than those of the plain genetic algorithm and the genetic algorithm often converges to local optima, as can be seen in the Table 5.11. It is interesting to watch the surrogate model based on a single regression tree hold itself well against the tough competition of ensemble methods such as AdaBoost R2 and stochastic gradient boosting, which performed the best among the tested surrogate model types.

By comparing values from the Table 5.11 and Table 5.12, it is obvious that after 2000 objective function evaluations, generation-based approach did beat individual-based approach for every single type of surrogate model tested. The convergence of the genetic algorithm is generally a little bit slower through the few initial generations with the surrogate model when utilizing generation-based evolution control, but more consistent throughout the whole run of the optimization algorithm.

Overall, it is hard to draw any conclusions about whether individual-based or generation-based approach is better based solely on the results from this benchmark function. Although the generation-based indeed performed better, individual-based approach offers a much bigger range of possible parameter settings which, properly tuned, could result in overall improvements in performance.

IndividualBased	WEC	RT	B	RF	SGB	ABR2
Mean	−544.26	−543.63	−539.51	−534.09	−537.64	−543.39
Deviation	8.21	9.73	13.83	18.12	14.84	9.44
Median	−546.64	−546.82	−544.44	−542.57	−545.28	−546.63
0.159 quantile	−547.43	−547.60	−547.13	−545.71	−547.21	−547.50
0.841 quantile	−544.40	−542.10	−538.35	−510.63	−525.97	−541.30

Table 5.11: Benchmark function by Valero et. al., individual-based control, summary after 2000 objective function evaluations. WEC - without evolution control, B - bagging, RF - random forest, SGB - stochastic gradient boosting, ABR2 - AdaBoost R2.

5.2.3 Performance on Benchmark Mixed Optimization Problem

The benchmark function in this section was proposed by Ocenasek and Schwarz ([31]) to benchmark estimation distribution algorithm for solving mixed continuous-discrete optimization problems. The optimization problem is stated as minimization problem of a function of n binary variables b_i and n continuous variables c_i

$$f(b_1, c_1, b_2, c_2, \ldots, b_n, c_n) = \sum_{i=1}^{n} g(b_i, c_i),$$

GenerationBased	WEC	RT	B	RF	SGB	ABR2
Mean	−544.26	−545.06	−543.68	−541.42	−546.25	−546.89
Deviation	8.21	9.48	11.47	14.37	2.56	0.79
Median	−546.64	−547.31	−546.95	−546.47	−546.91	−547.09
0.159 quantile	−547.43	−547.55	−547.54	−547.43	−547.50	−547.58
0.841 quantile	−544.40	−546.06	−545.06	−543.23	−545.54	−546.03

Table 5.12: Benchmark function by Valero et. al., generation-based control, summary after 2000 objective function evaluations. WEC - without evolution control, B - bagging, RF - random forest, SGB - stochastic gradient boosting, ABR2 - AdaBoost R2.

where

$$g(b_i, c_i) = \begin{cases} 0.8 - 0.5c_i & \text{if } b_i = 0 \\ 0.8 - c_i & \text{if } b_i = 1 \text{ and } c_i \leq 0.8 \\ 5(c_i - 0.8) & \text{if } b_i = 1 \text{ and } c_i > 0.8. \end{cases}$$

Although the benchmark function contains categorical variables, it can be represented by a single polytope equivalence class. As linear constraints are left unspecified by Ocenasek and Schwarz in the cited article, in the experiments it was decided to restrict values of each continuous variable c_i to the interval $[-1, 1]$. Moreover, $n = 10$ was used.

It can be seen easily that subject to the presented linear constraints, the function attains its minimum of 0 when $b_i = 1$ and $c_i = 0.8$ for $i = 1, \ldots, n$.

Three genetic operators were applied to create the offspring - 50% of offspring individuals were created using the discrete crossover with mutation, 25% using the mutation on all variables and the remaining 25% of offspring individuals using the continuous ball mutation.

The overview of results obtained after 2000 objective function evaluations with the methodology established in Section 5.2.1 are summarized in Table 5.13 and Table 5.14.

Detailed graphs showing performance of individual-based evolution control are presented in Figures 5.10-5.13, while the performance of generation-based control is depicted in Figures 5.14-5.17. Each combination of surrogate model and evolution control type is shown in exactly two figures, both figures comparing the obtained results to the results obtained without evolution control. The difference between the two figures lies in the way the results are presented - the first figure shows the mean plus minus standard deviation of the objective function values obtained in 50 runs of the genetic algorithm. Second figure, on the other hand, shows the median, 0.159 and 0.841 quantiles of the objective function values obtained in 50 runs of the genetic algorithm.

In individual-based control, all tested types of surrogate models help to considerably accelerate the convergence of the evolutionary algorithm with the exception of the surrogate model based on a single regression tree, which performs unarguably badly. From the ensemble methods, stochastic gradient boosting and AdaBoost R2 seem to work the best, bagging comes in the middle and random forest perform the worst.

Judging by mean/median values after 2000 objective function evaluations, generation-based control always performs better than individual-based control

with all tested types of surrogate models, exactly as with the benchmark function by Valero et. al. Ensemble methods perform better than a single regression tree model, but aside from that, the performance of ensemble methods is pretty much balanced, with no method being considerably better than others. For example, stochastic gradient boosting after 2000 objective function evaluations reaches median value of 0.87, which is approximately half of the value 1.79 obtained without evolution control, a nice improvement indeed.

It is possible that the performance of individual-based control could be improved by choosing smaller offspring population multiplier than 10; without clustering strategy to maintain diversity in the population, multiplier of 10 may just create too much selection pressure.

IndividualBased	WEC	RT	B	RF	SGB	ABR2
Mean	1.78	2.60	1.25	1.73	1.07	1.02
Deviation	0.33	0.52	0.51	0.51	0.32	0.30
Median	1.79	2.58	1.17	1.70	1.08	0.99
0.159 quantile	1.47	2.04	0.75	1.22	0.77	0.76
0.841 quantile	2.14	3.11	1.79	2.22	1.40	1.31

Table 5.13: Benchmark function by Ocenasek and Schwarz, individual-based control, summary after 2000 objective function evaluations. WEC - without evolution control, B - bagging, RF - random forest, SGB - stochastic gradient boosting, ABR2 - AdaBoostR2.

GenerationBased	WEC	RT	B	RF	SGB	ABR2
Mean	1.78	1.34	0.92	0.96	0.87	0.91
Deviation	0.33	0.31	0.25	0.24	0.18	0.23
Median	1.79	1.34	0.91	0.92	0.87	0.89
0.159 quantile	1.47	1.02	0.69	0.73	0.70	0.70
0.841 quantile	2.14	1.64	1.21	1.26	1.06	1.18

Table 5.14: Benchmark function by Ocenasek and Schwarz, generation-based control, summary after 2000 objective function evaluations. WEC - without evolution control, B - bagging, RF - random forest, SGB - stochastic gradient boosting, ABR2 - AdaBoostR2.

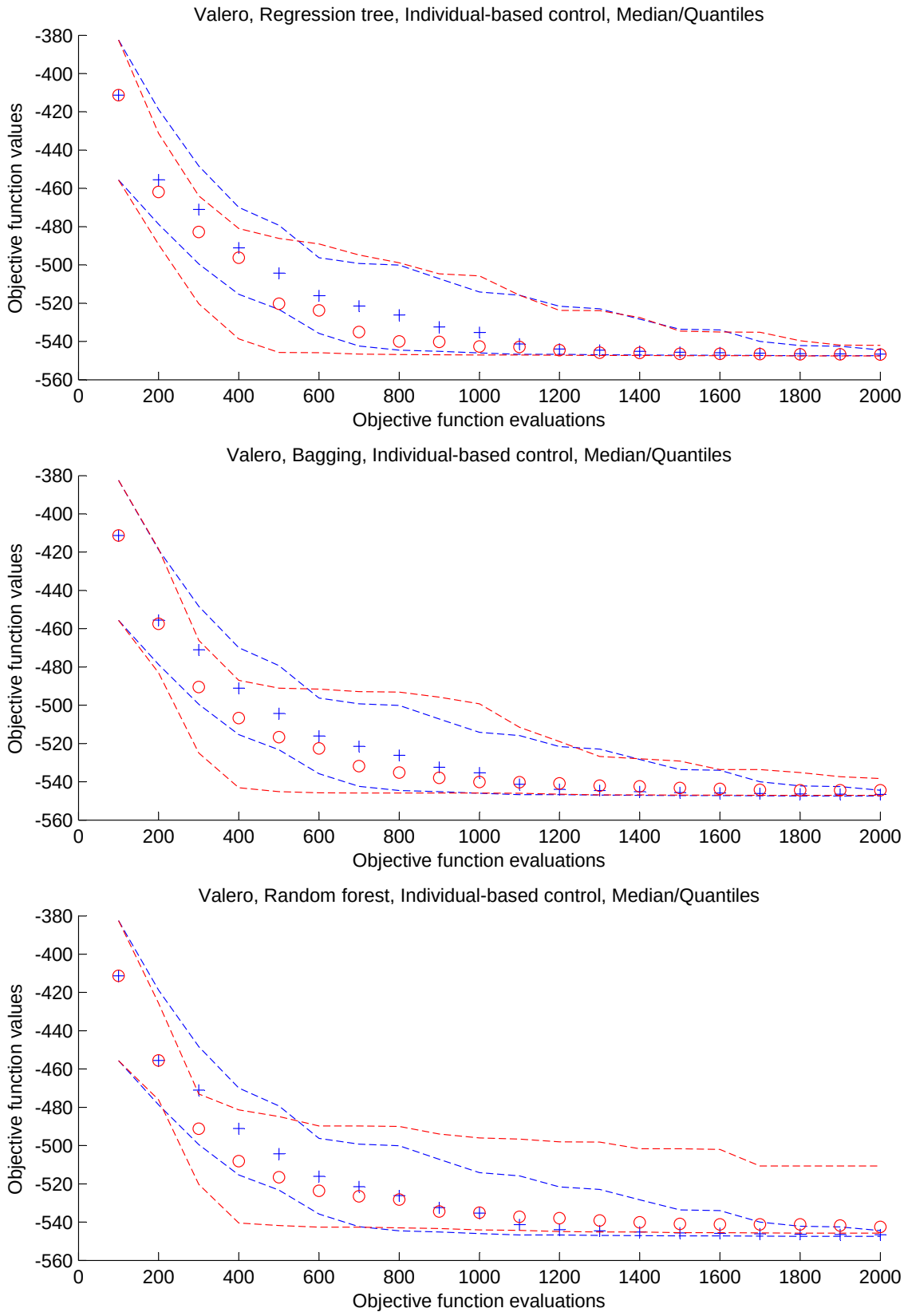


Figure 5.2: Benchmark function by Valero et. al, individual-based control, median/quantiles. Performance with the surrogate model is depicted in red, while the performance without the surrogate model is depicted in blue.

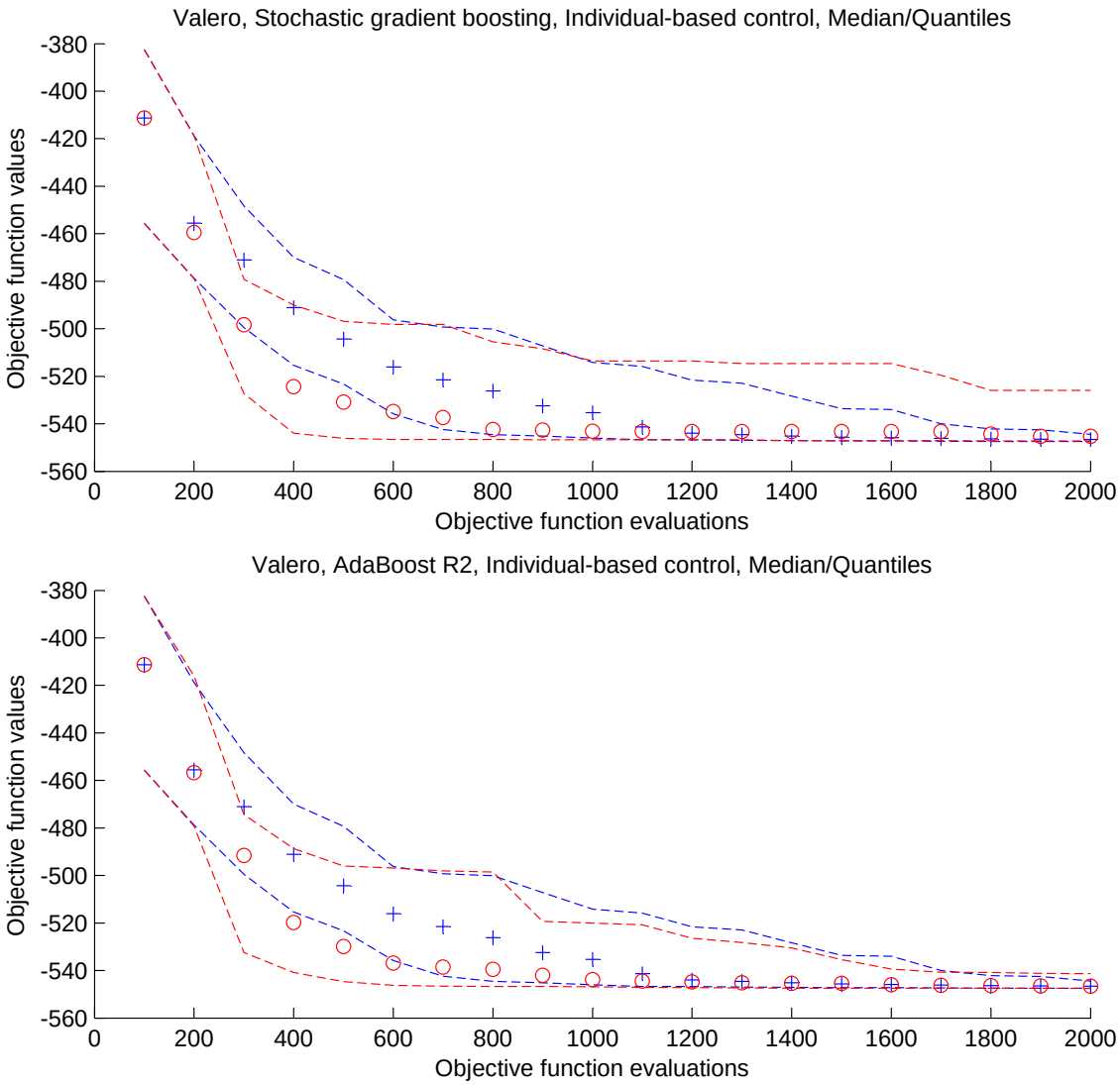


Figure 5.3: Benchmark function by Valero et. al, individual-based control, median/quantiles. Performance with the surrogate model is depicted in red, while the performance without the surrogate model is depicted in blue.

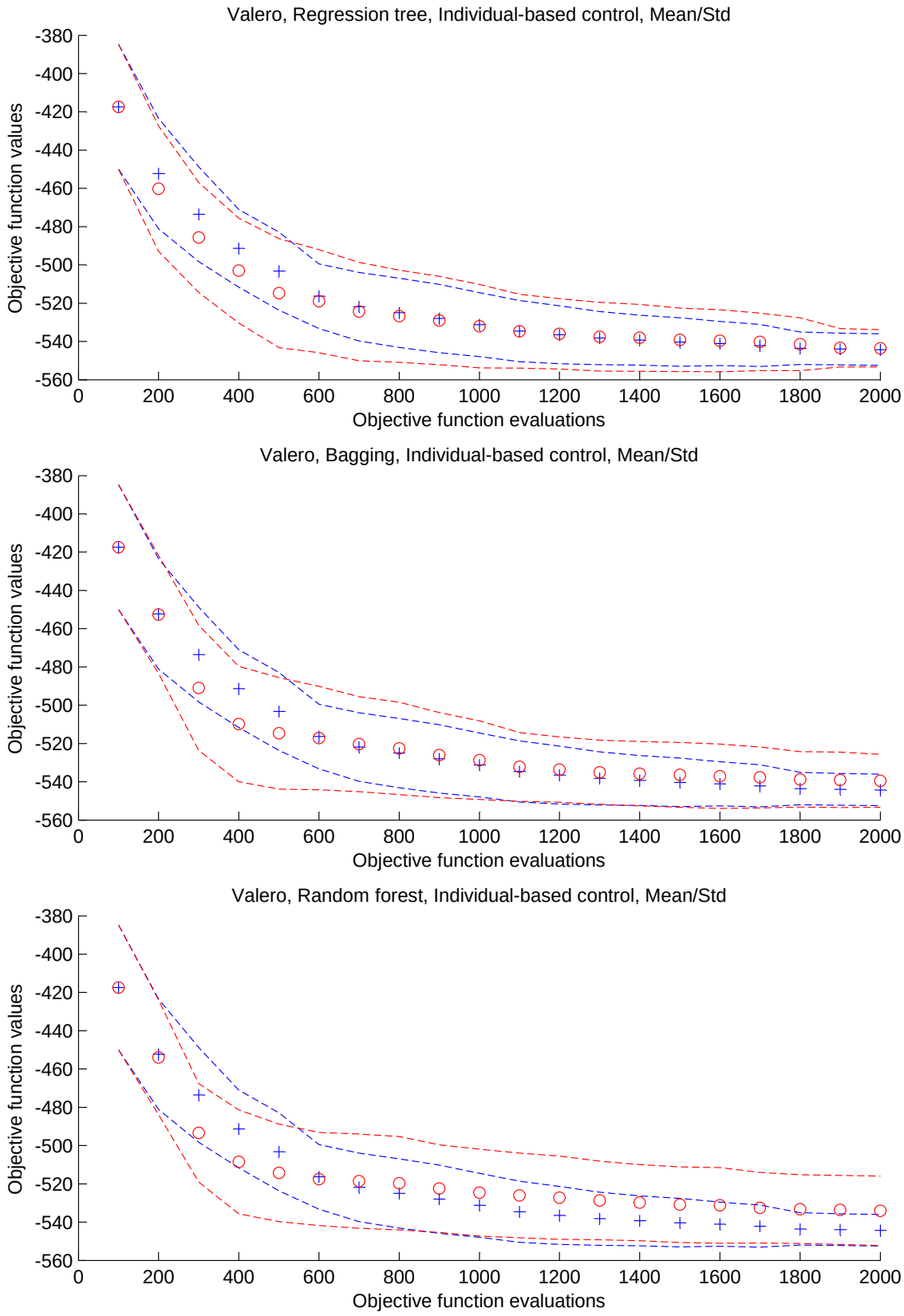


Figure 5.4: Benchmark function by Valero et. al, individual-based control, mean/deviation. Performance with the surrogate model is depicted in red, while the performance without the surrogate model is depicted in blue.

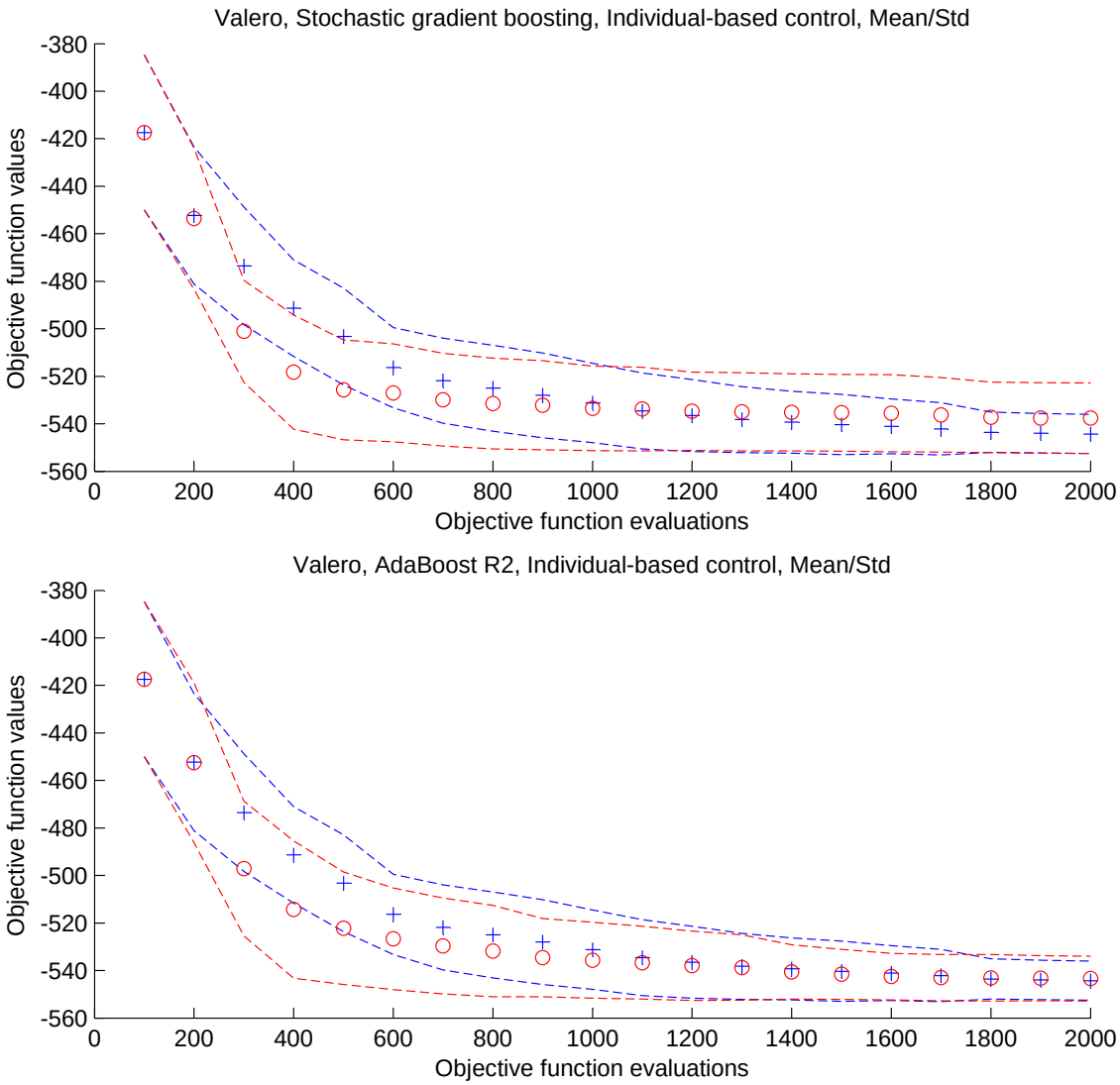


Figure 5.5: Benchmark function by Valero et. al, individual-based control, mean/deviation. Performance with the surrogate model is depicted in red, while the performance without the surrogate model is depicted in blue.

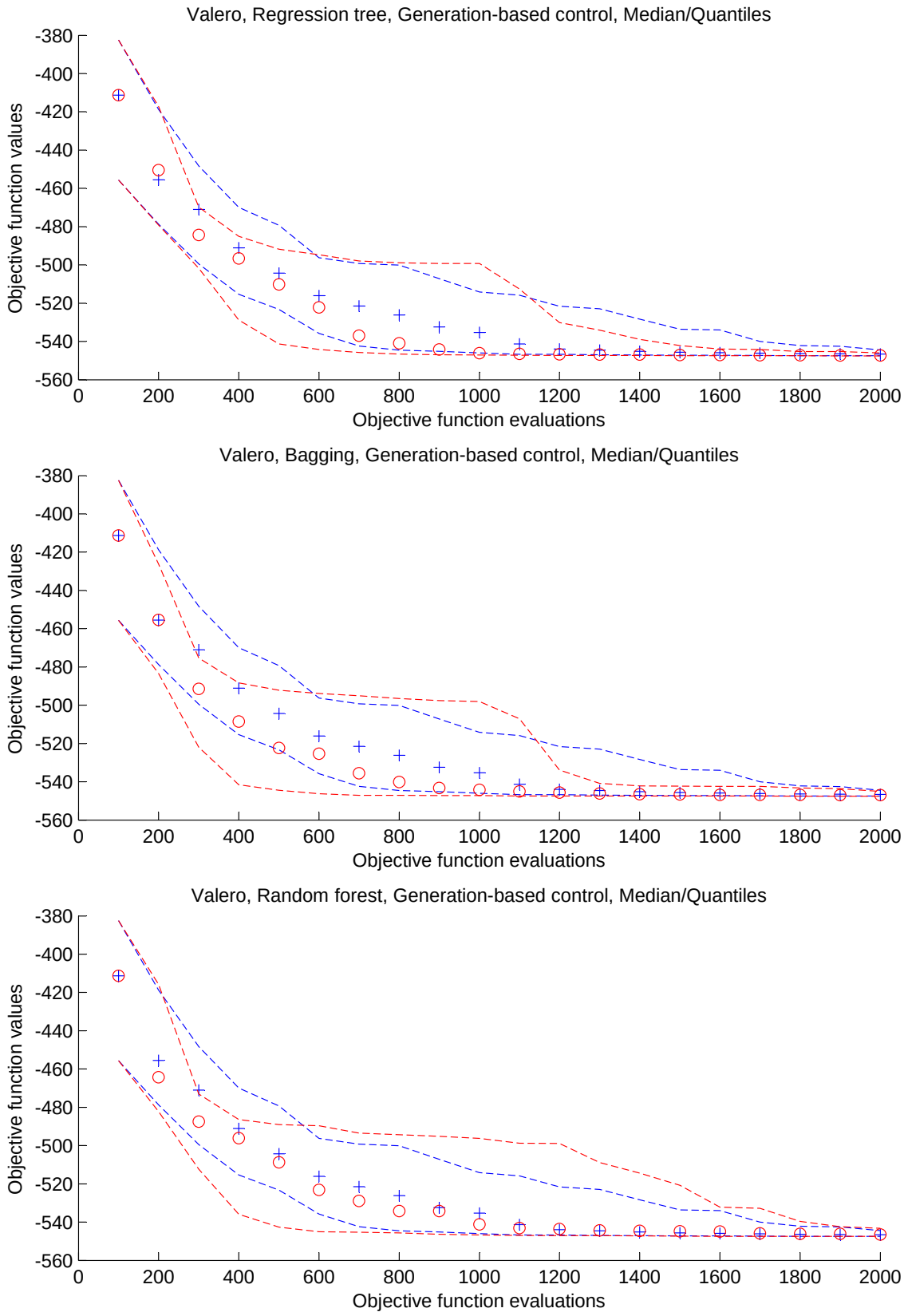


Figure 5.6: Benchmark function by Valero et. al, generation-based control, median/quantiles. Performance with the surrogate model is depicted in red, while the performance without the surrogate model is depicted in blue.

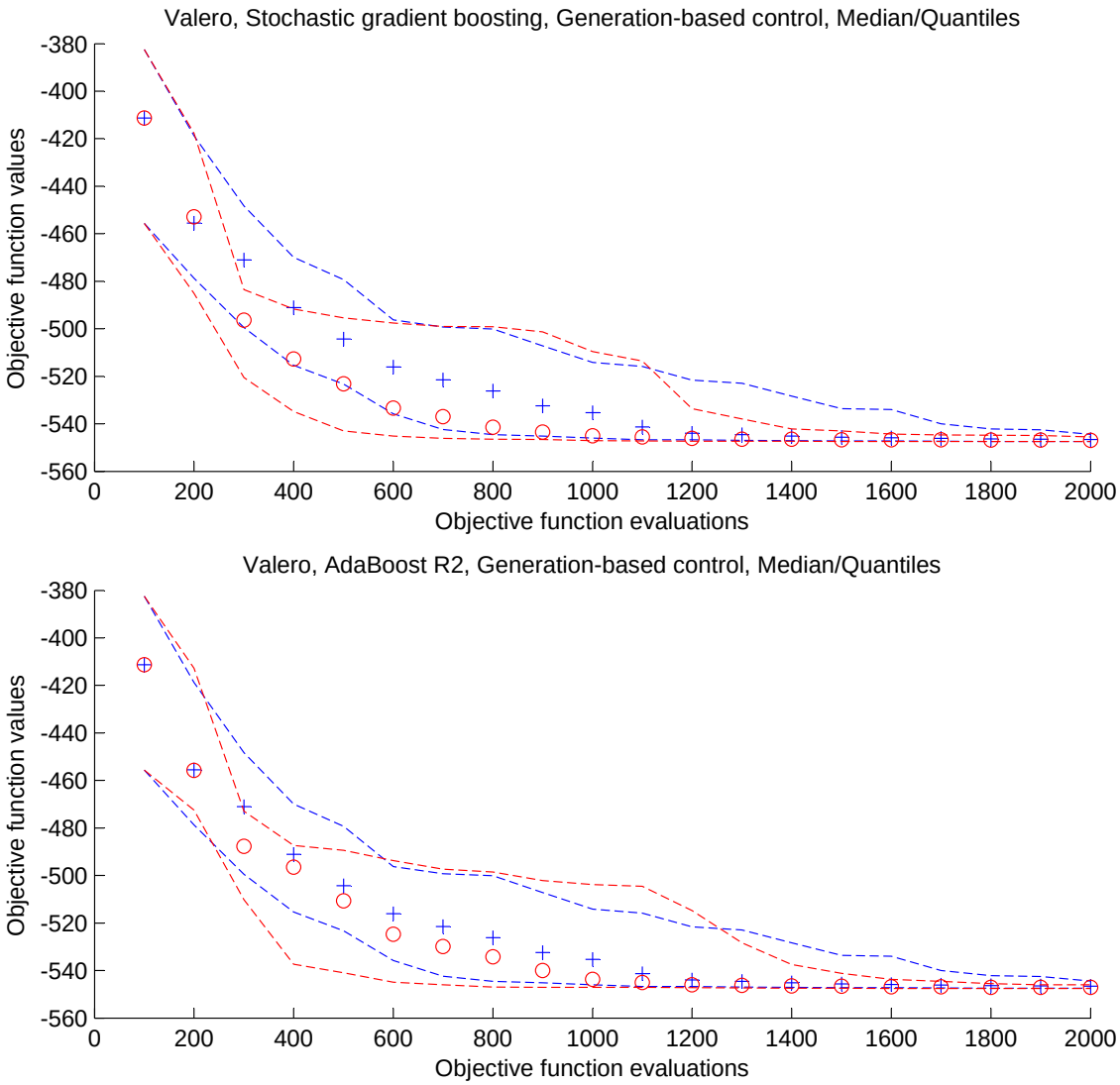


Figure 5.7: Benchmark function by Valero et. al, generation-based control, median/quantiles. Performance with the surrogate model is depicted in red, while the performance without the surrogate model is depicted in blue.

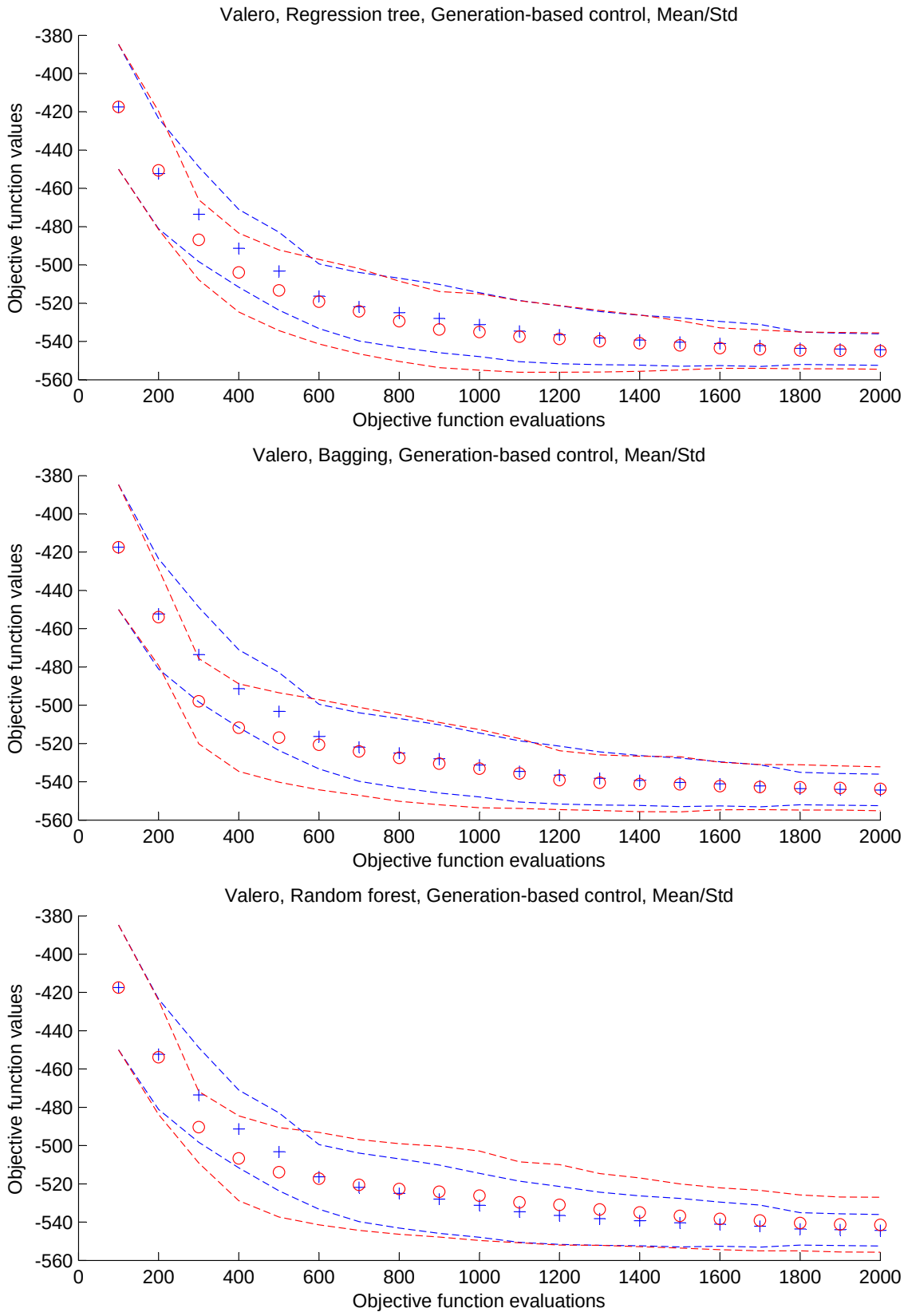


Figure 5.8: Benchmark function by Valero et. al, generation-based control, mean/deviation. Performance with the surrogate model is depicted in red, while the performance without the surrogate model is depicted in blue.

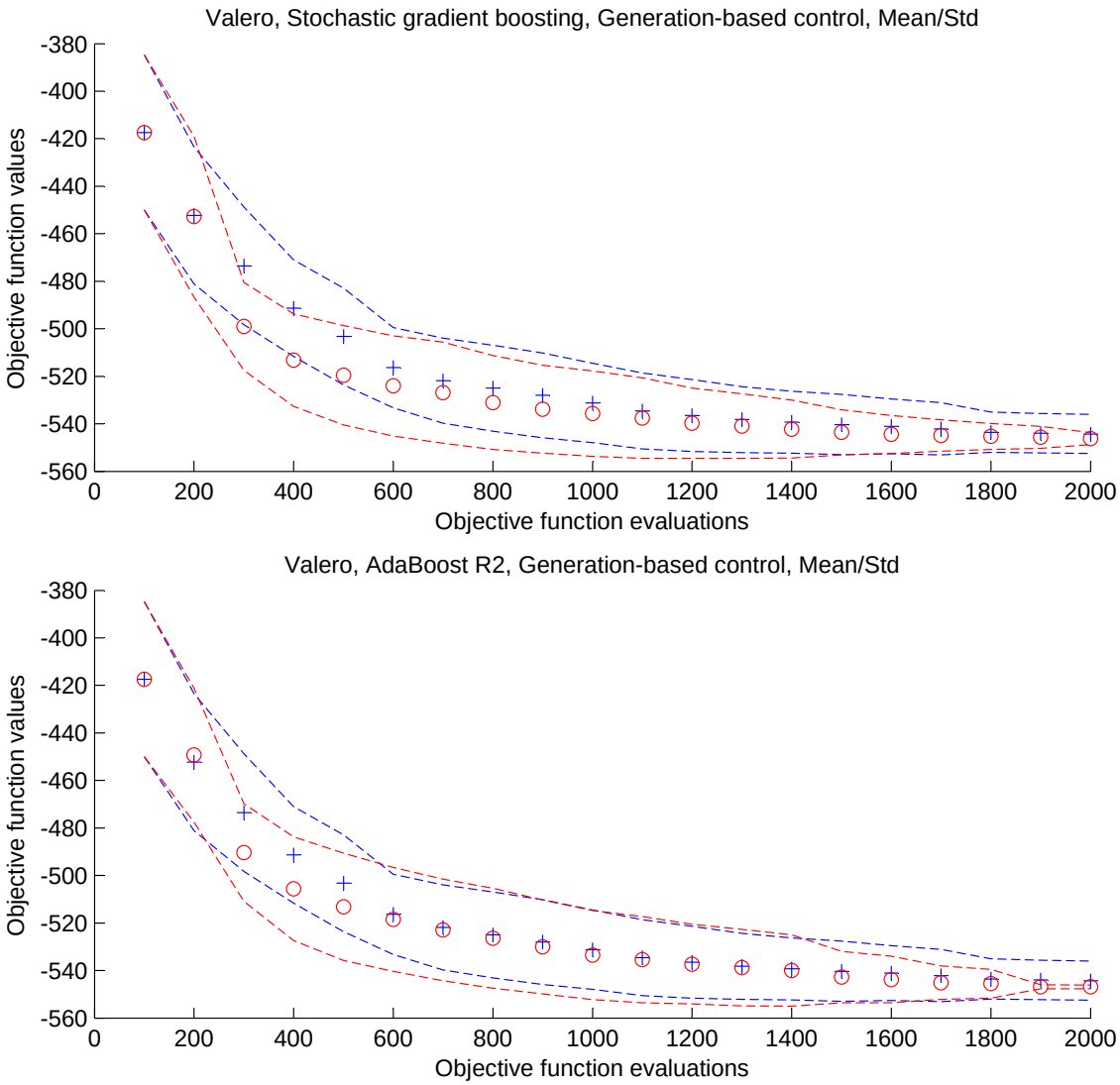


Figure 5.9: Benchmark function by Valero et. al, generation-based control, mean/deviation. Performance with the surrogate model is depicted in red, while the performance without the surrogate model is depicted in blue.

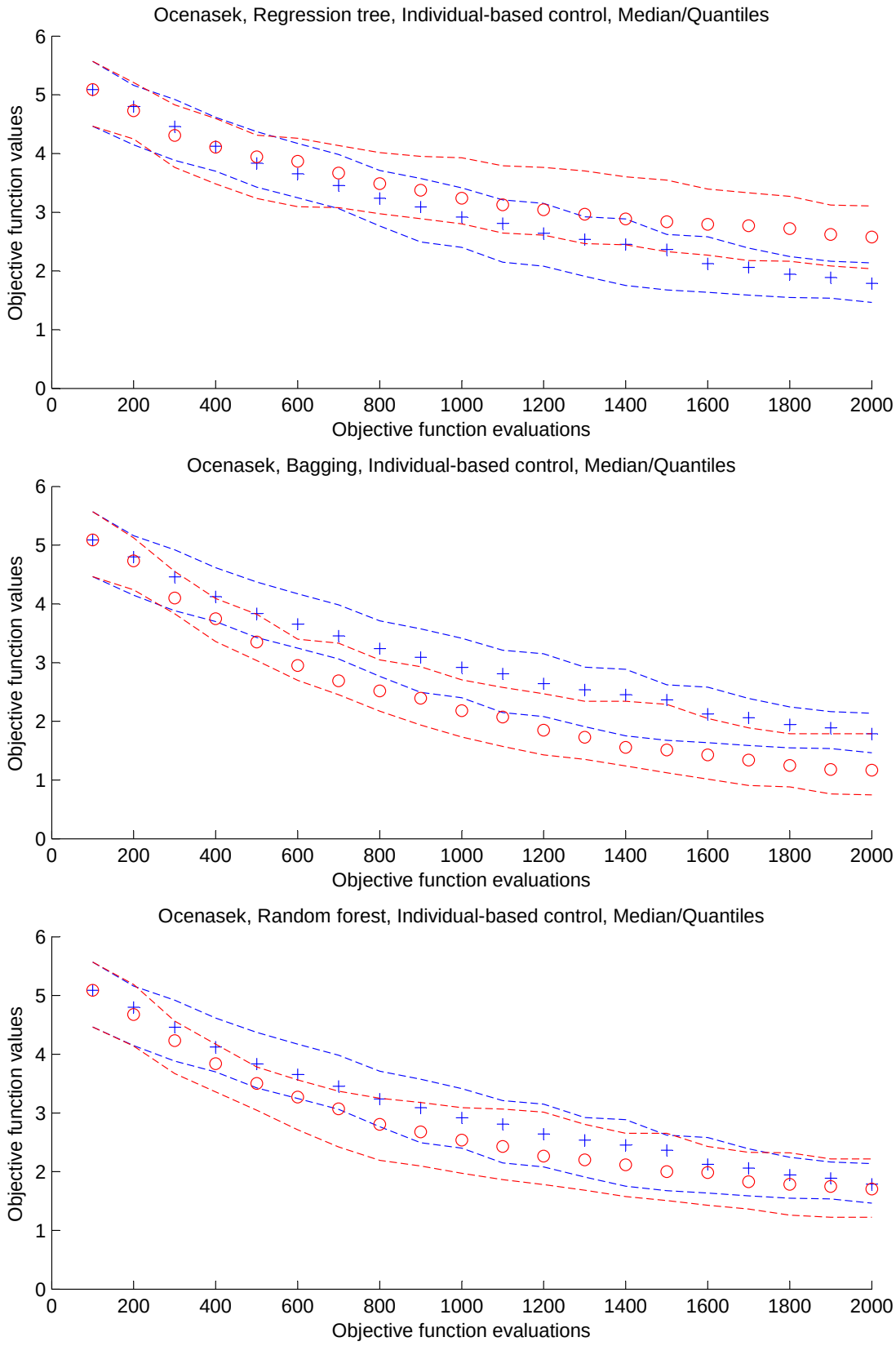


Figure 5.10: Benchmark function by Ocenasek and Schwarz, individual-based control, median/quantiles. Performance with the surrogate model is depicted in red, while the performance without the surrogate model is depicted in blue.

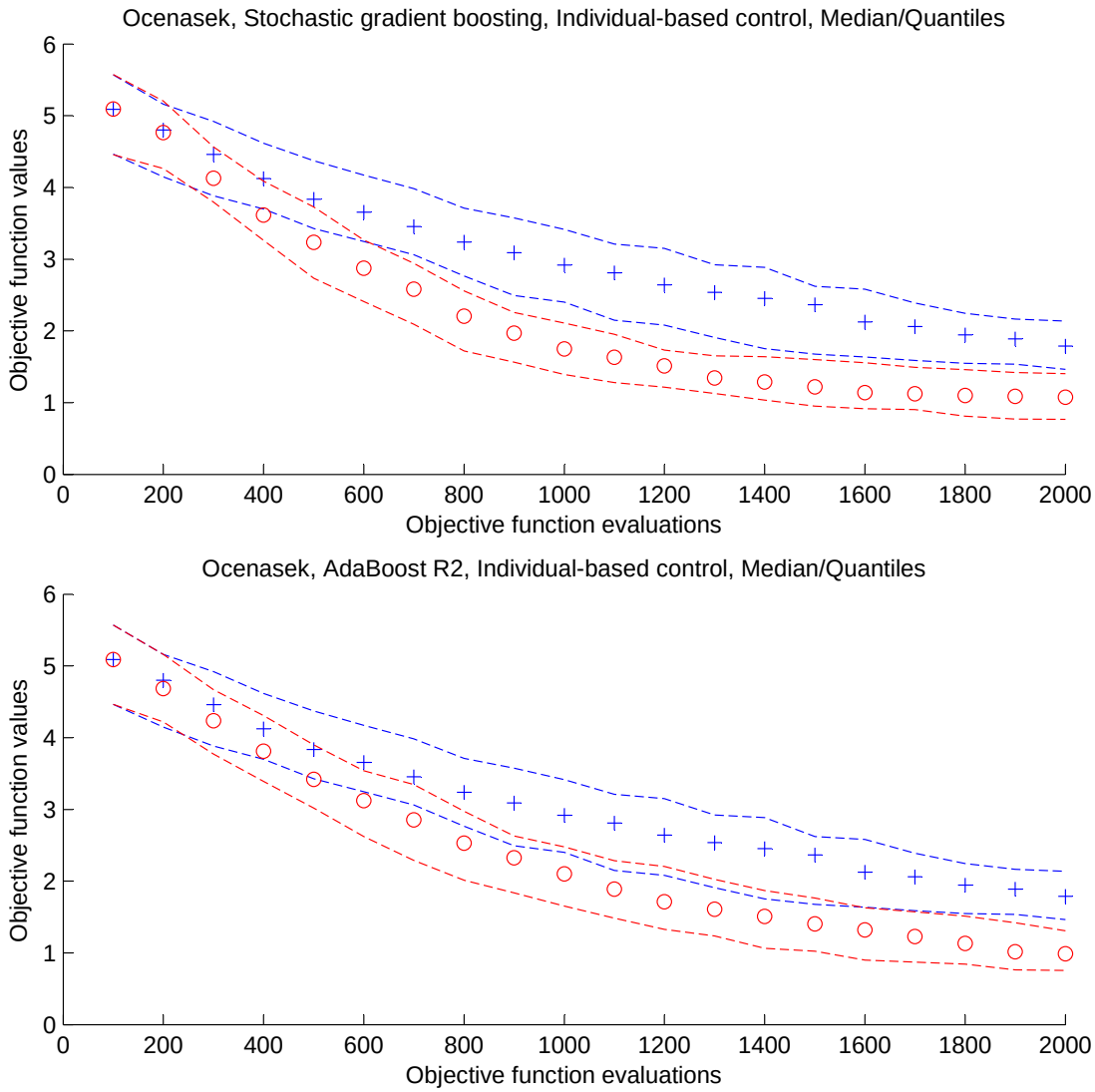


Figure 5.11: Benchmark function by Ocenasek and Schwarz, individual-based control, median/quantiles. Performance with the surrogate model is depicted in red, while the performance without the surrogate model is depicted in blue.

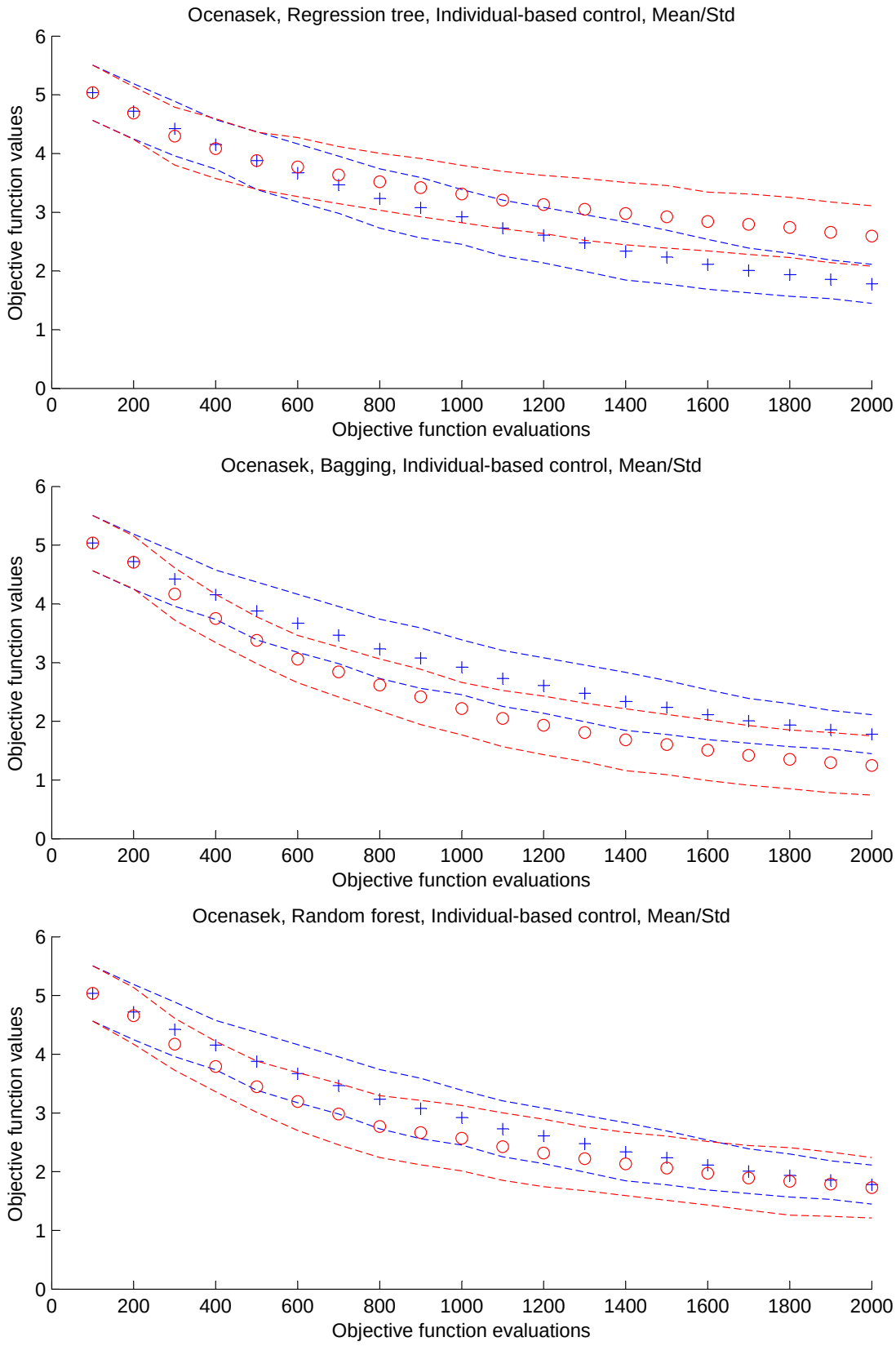


Figure 5.12: Benchmark function by Ocenasek and Schwarz, individual-based control, mean/deviation. Performance with the surrogate model is depicted in red, while the performance without the surrogate model is depicted in blue.

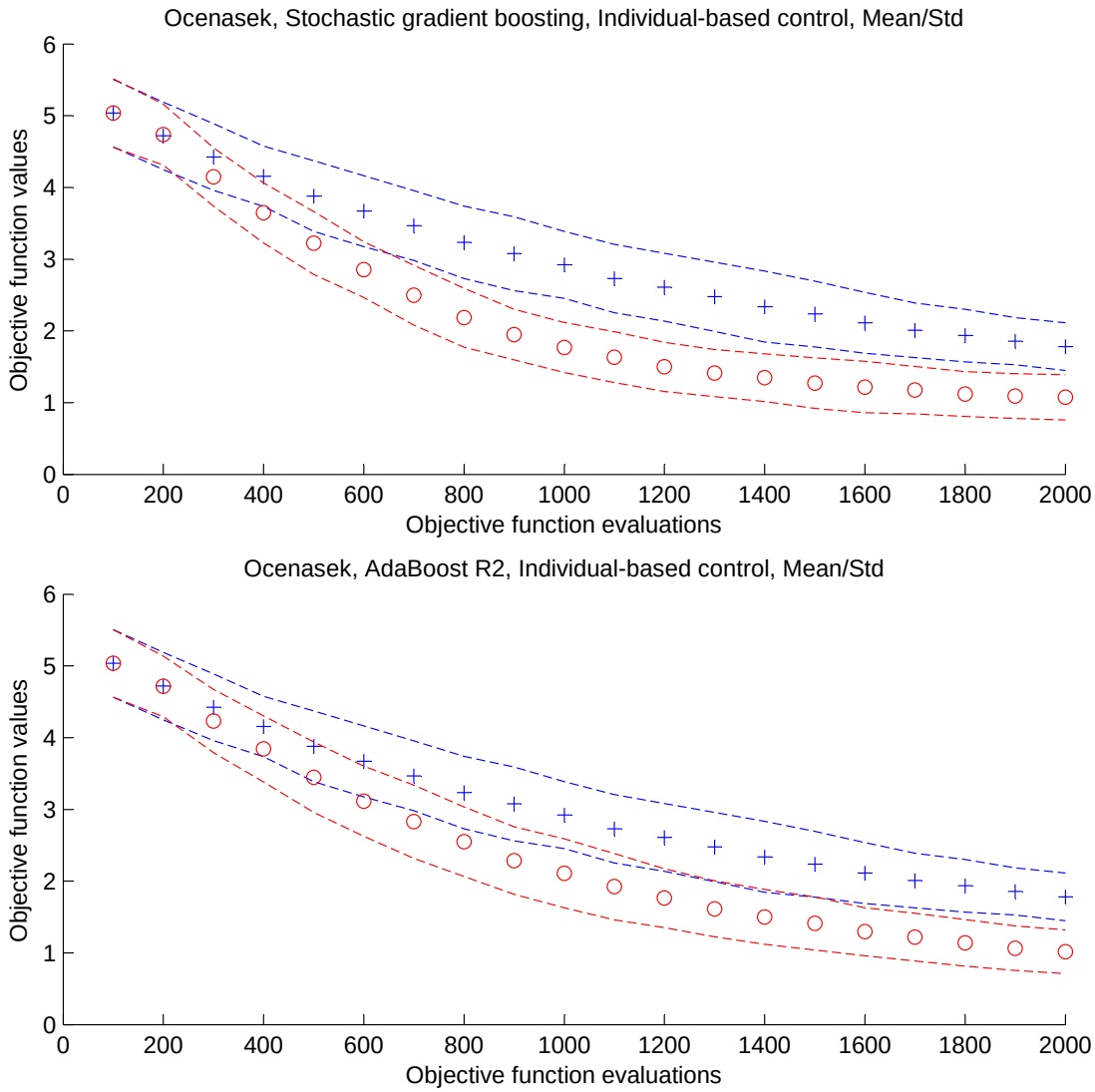


Figure 5.13: Benchmark function by Ocenasek and Schwarz, individual-based control, mean/deviation. Performance with the surrogate model is depicted in red, while the performance without the surrogate model is depicted in blue.

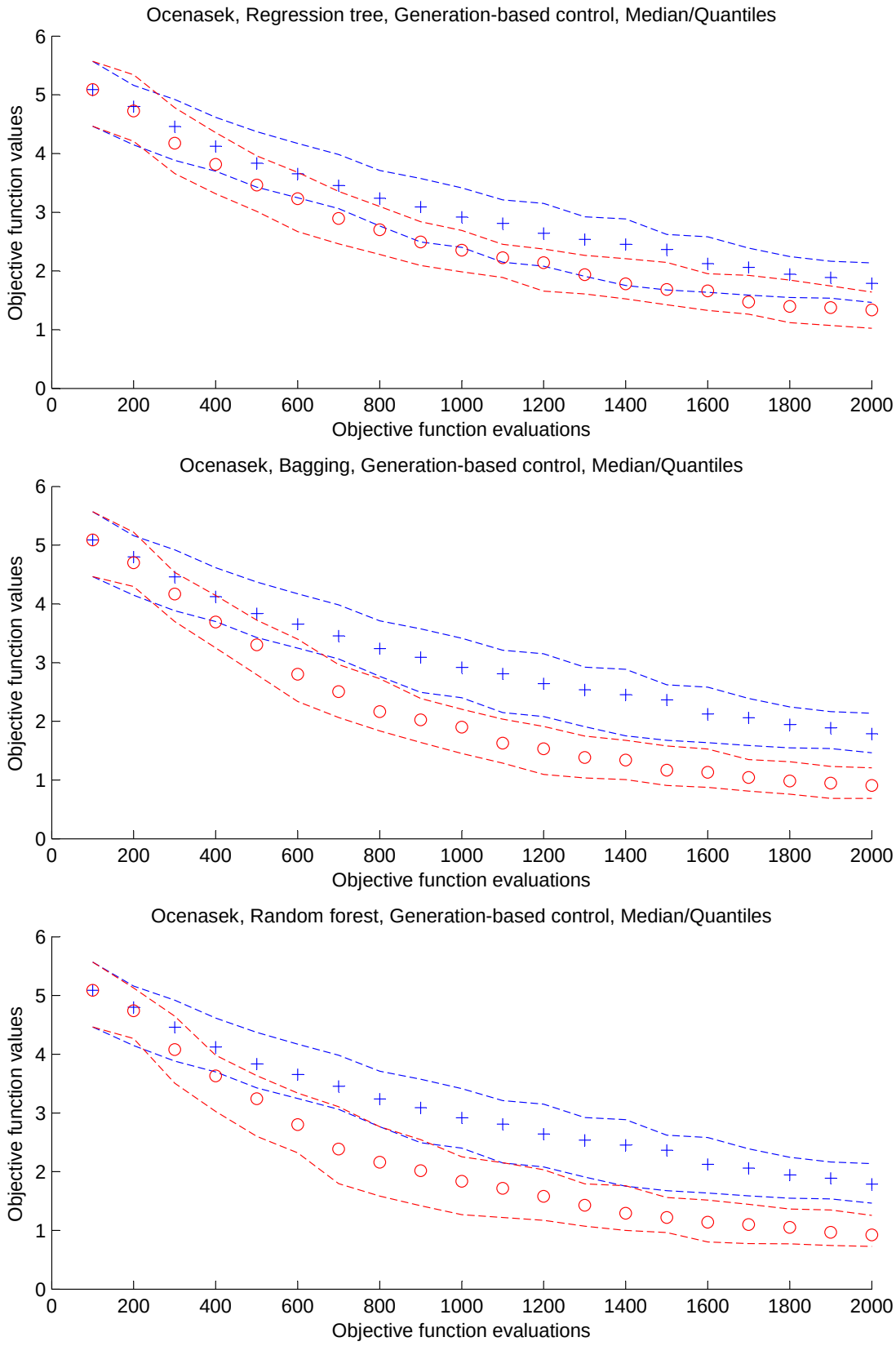


Figure 5.14: Benchmark function by Ocenasek and Schwarz, generation-based control, median/quantiles. Performance with the surrogate model is depicted in red, while the performance without the surrogate model is depicted in blue.

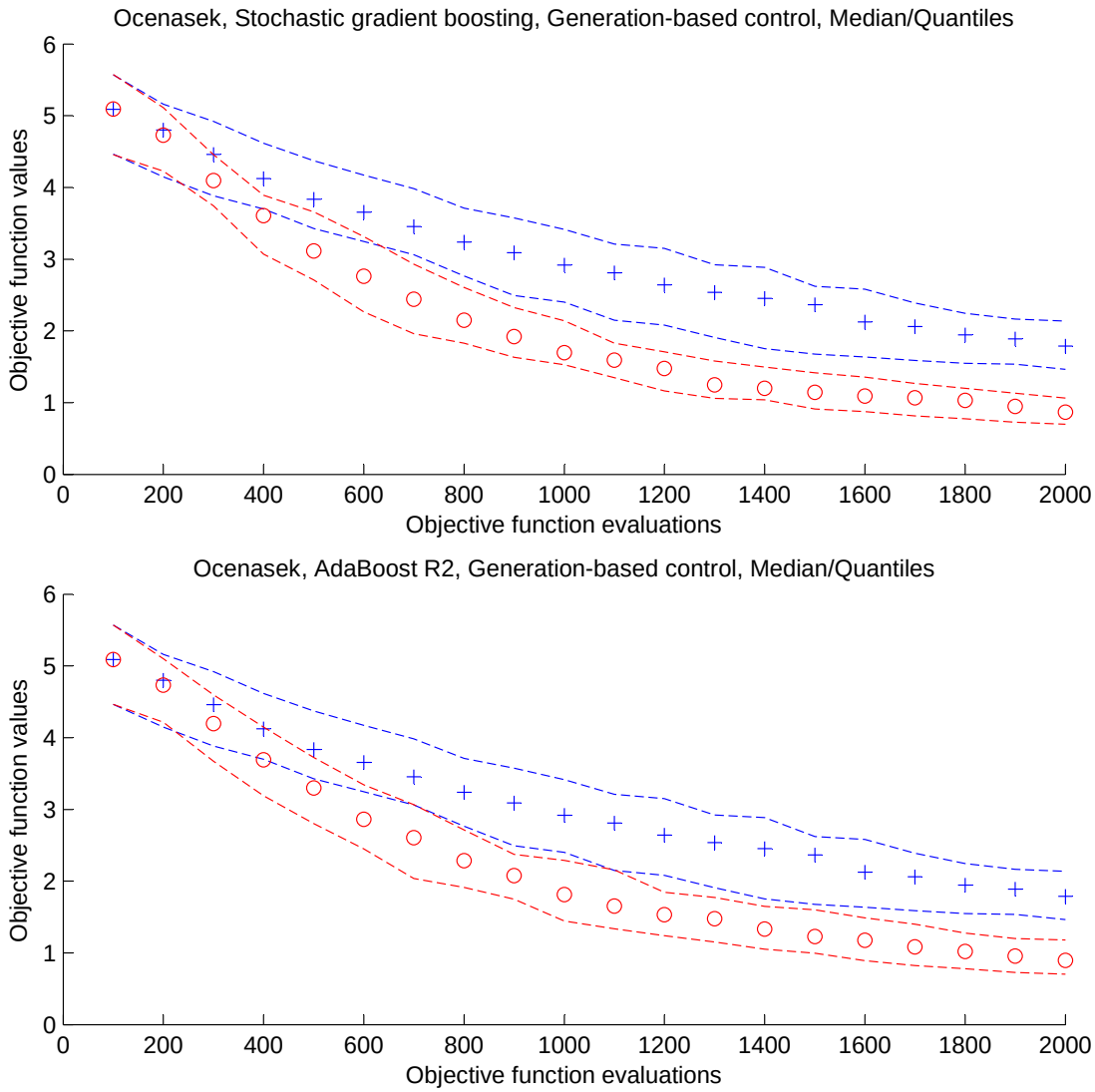


Figure 5.15: Benchmark function by Ocenasek and Schwarz, generation-based control, median/quantiles. Performance with the surrogate model is depicted in red, while the performance without the surrogate model is depicted in blue.

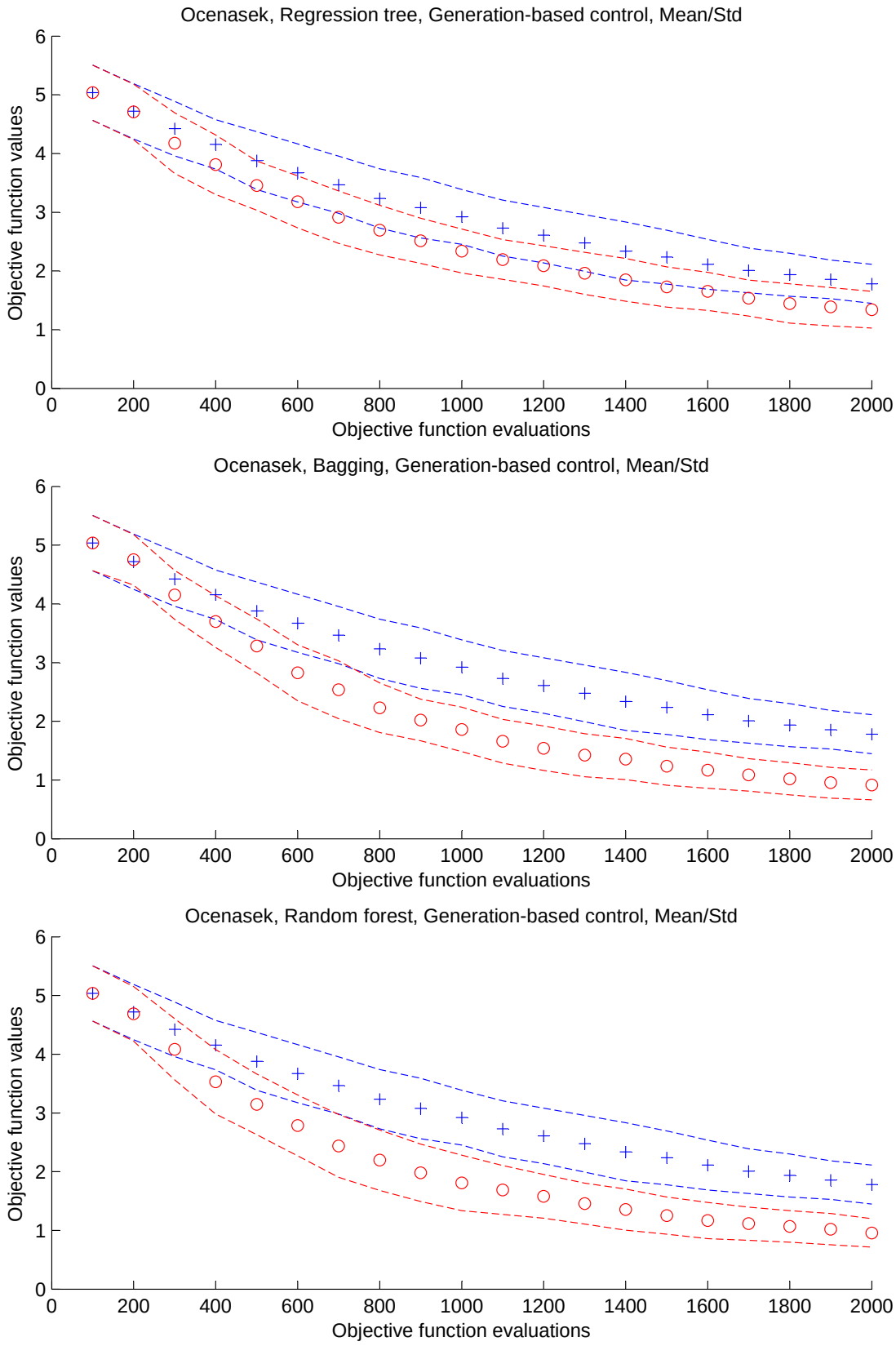


Figure 5.16: Benchmark function by Ocenasek and Schwarz, generation-based control, mean/deviation. Performance with the surrogate model is depicted in red, while the performance without the surrogate model is depicted in blue.

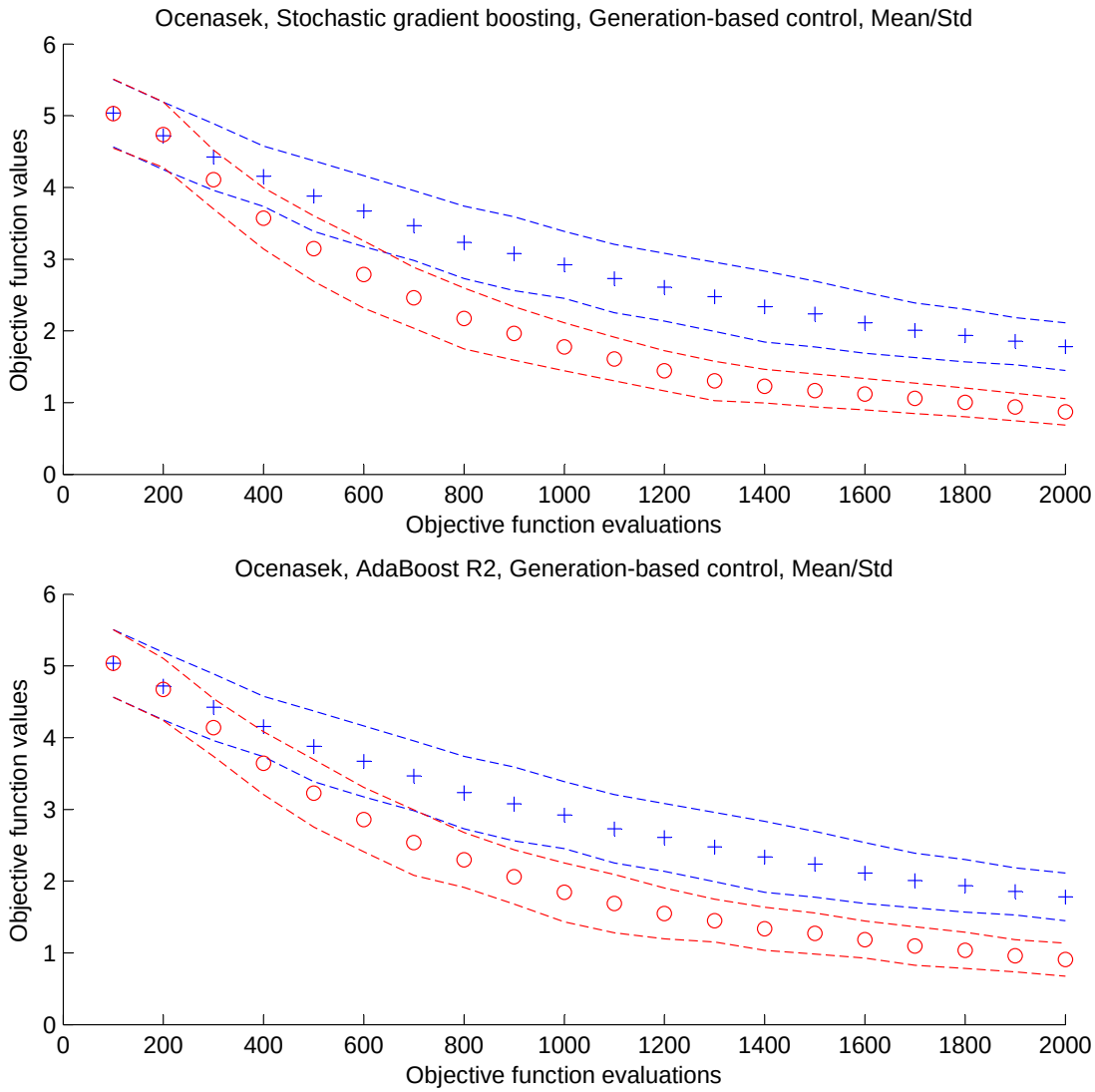


Figure 5.17: Benchmark function by Ocenasek and Schwarz, generation-based control, mean/deviation. Performance with the surrogate model is depicted in red, while the performance without the surrogate model is depicted in blue.

6. Conclusion

6.1 Future Work

The proposed research could be extended in many ways, including:

- **Utilizing the proposed genetic algorithm in practice** - The prototype implementation of the proposed methods has only been tested on benchmark functions together with a very narrow set of possible parameter settings. In future, the algorithm should be tuned on more benchmark problems and, finally, employed in practice.
- **Multiple surrogate models** - Evolutionary optimization algorithms are not restricted to use a single surrogate model, they may actually profit from using multiple surrogate models. For example, it is quite likely that models based on artificial neural networks will perform better when applied to sets of solutions that share the same values of categorical variables and only differ in values of continuous variables. Therefore, individuals obtained by certain genetic operators (such as mutations of values of continuous variables) could be evaluated using a more accurate local model, instead of the global one based on decision trees.
- **Hybrid optimization** - while evolutionary algorithms are good for finding high quality solutions to the solved problem, other optimization methods may be utilized to locally improve the solutions obtained by the genetic algorithm. For example, gradient-based optimization methods could probably effectively improve continuous parts of promising solutions.
- **Online learning** - the proposed surrogate model is retrained each time new data are evaluated by the objective function. The cost of retraining the surrogate model could be minimized by using online learning methods ([11]), which only update the existing model based on the newly obtained data samples. While for testing purposes this may represent a significant speedup of the algorithm, for practical purposes online learning is of a little help, since we usually assume that the training of the surrogate model is always much cheaper than the evaluation of an expensive empirical objective function.

6.2 Summary

Evolutionary optimization of empirical objective functions can be considerably accelerated by using a surrogate model in place of the objective function. To contribute to the topic of surrogate modelling in evolutionary optimization, the main goal of this thesis was to propose a surrogate model based on regression trees and their generalizations and methods of utilizing it in evolutionary optimization of objective functions of both continuous and categorical variables .

In Chapter 2, we started by reviewing the some of the most widely recognized regression methods based on regression trees and their ensembles, focusing on the

most important ideas behind each method. We also paid attention to basic ideas behind evolutionary optimization and the role of surrogate modelling in it.

Taking advantage of this knowledge, genetic algorithm utilizing a surrogate model based on regression trees and their generalizations has been proposed in Section 3.1, including pseudocode of the body of the algorithm, with each part of the algorithm being presented in a very high level of detail. Design of certain parts of the algorithm has been motivated by a class of optimization problems encountered in catalysis

Prototype implementation of the algorithm has been carried out in *MATLAB* environment and thoroughly tested. In the first phase of testing, tree-based regression methods and the proposed surrogate model were tested using both real-world data sets and artificial benchmark functions. Ensemble models have been shown to significantly outperform single regression trees.

In the second phase of benchmarking, the genetic algorithm utilizing the proposed surrogate model was tested as a whole on two benchmark functions. In both cases, the proposed surrogate model performed well in both individual-based and generation-based control. Interestingly, generation-based control consistently outperformed individual-based control under the adopted testing methodology. When testing various types of surrogate models, especially stochastic gradient boosting and AdaBoost R2 impressed.

Bibliography

[1] H.G. Beyer and H.P. Schwefel. Evolution strategies–A comprehensive introduction. *Natural computing*, 1(1):3–52, 2002.

[2] L. Breiman. *Classification and regression trees*. Chapman & Hall/CRC, 1984.

[3] L. Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[4] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[5] D. Buche, N.N. Schraudolph, and P. Koumoutsakos. Accelerating evolutionary algorithms using fitness function models. In *Proc. GECCO Workshop Learn., Adapt. Approx. Evol. Comput*, pages 166–169. Citeseer, 2003.

[6] L. Bull. On model-based evolutionary computation. *Soft Computing-A Fusion of Foundations, Methodologies and Applications*, 3(2):76–82, 1999.

[7] H. Drucker. Improving regressors using boosting techniques. In *Machine learning-international workshop then conference*, pages 107–115. Citeseer, 1997.

[8] A.E. Eiben and J.E. Smith. *Introduction to evolutionary computing*. Springer Verlag, 2003.

[9] M.A. El-Beltagy and AJ Keane. Metalmodeling techniques for evolutionary optimization of computationally expensive problems: promises and limitations. *Proceedings of the Genetic and Evolutionary Computation Conference. Genetic and Evolutionary Computation Conference*, pages 196–203, 1999.

[10] M. Emmerich, A. Giotis, M. Ozdemir, T. Back, and K. Giannakoglou. Metamodel—Assisted Evolution Strategies. *Parallel Problem Solving from Nature—PPSN VII*, pages 361–370, 2002.

[11] A. Fern and R. Givan. Online ensemble learning: An empirical study. *Machine Learning*, 53(1):71–109, 2003.

[12] L.J. Fogel. Intelligence through simulated evolution: Four decades of evolutionary programming, 1999.

[13] L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial intelligence through simulated evolution*. John Wiley, 1966.

[14] Y. Freund. Boosting a weak learning algorithm by majority. *Information and computation*, 121(2):256–285, 1995.

[15] Y. Freund and R. Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *Computational learning theory*, pages 23–37. Springer, 1995.

[16] J.H. Friedman. Multivariate adaptive regression splines. *The annals of statistics*, 19(1):1–67, 1991.

[17] J.H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, pages 1189–1232, 2001.

[18] D.E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-wesley, 1989.

[19] T. Hastie, R. Tibshirani, and J.H. Friedman. *The elements of statistical learning: data mining, inference, and prediction*. Springer Verlag, 2001.

[20] R.L. Haupt, S.E. Haupt, and J. Wiley. *Practical genetic algorithms*. Wiley Online Library, 1998.

[21] R. Jin, W. Chen, and T.W. Simpson. Comparative studies of metamodelling techniques under multiple modelling criteria. *Structural and Multidisciplinary Optimization*, 23(1):1–13, 2001.

[22] Y. Jin. A comprehensive survey of fitness approximation in evolutionary computation. *Soft Computing-A Fusion of Foundations, Methodologies and Applications*, 9(1):3–12, 2005.

[23] Y. Jin, M. Olhofer, and B. Sendhoff. On evolutionary optimization with approximate fitness functions. In *Proceedings of the genetic and evolutionary computation conference*, pages 786–793, 2000.

[24] Y. Jin and B. Sendhoff. Reducing fitness evaluations using clustering techniques and neural network ensembles. In *Genetic and Evolutionary Computation–GECCO 2004*, pages 688–699. Springer, 2004.

[25] R. Kannan and H. Narayanan. Random walks on polytopes and an affine interior point method for linear programming. In *Proceedings of the 41st annual ACM symposium on Theory of computing*, pages 561–570. ACM, 2009.

[26] J. Koza and R. Poli. Genetic programming. *Search Methodologies*, pages 127–164, 2005.

[27] J.R. Koza. Genetic programming: On the programming of computers by natural selection, 1992.

[28] R.M. Lewis, A. Shepherd, and V. Torczon. Implementing generating set search methods for linearly constrained minimization. *SIAM journal on scientific computing*, 29(6):2507–2530, 2007.

[29] R.M. Lewis, V. Torczon, Institute for Computer Applications in Science, and Engineering. *Pattern search methods for linearly constrained minimization*, volume 206904. Citeseer, 1998.

[30] S. Mohmel, N. Steinfeldt, S. Engelschalt, M. Holena, S. Kolf, M. Baerns, U. Dingerdissen, D. Wolf, R. Weber, and M. Bewersdorf. New catalytic materials for the high-temperature synthesis of hydrocyanic acid from methane and ammonia by high-throughput approach. *Applied Catalysis A: General*, 334(1-2):73–83, 2008.

[31] J. Ocenasek and J. Schwarz. Estimation distribution algorithm for mixed continuous-discrete optimization problems. *Intelligent technologies: theory and applications: new trends in intelligent technologies*, 76:227, 2002.

[32] A. Ratle. Accelerating the convergence of evolutionary algorithms by fitness landscape approximation. In *Parallel Problem Solving from Nature—PPSN V*, pages 87–96. Springer, 1998.

[33] R.E. Schapire. The boosting approach to machine learning: An overview. *Lecture Notes in Statistics*, pages 149–172, 2003.

[34] H.P. Schwefel. Evolution strategies: A family of non-linear optimization techniques based on imitating some principles of organic evolution. *Annals of Operations Research*, 1(2):165–167, 1984.

[35] H. Ulmer, F. Streichert, and A. Zell. Evolution strategies with controlled model assistance. In *Evolutionary Computation, 2004. CEC2004. Congress on*, volume 2, pages 1569–1576. IEEE, 2004.

[36] H. Ulmer, F. Streichert, and A. Zell. Model assisted evolution strategies. *Knowledge Incorporation in Evolutionary Computation*, 333, 2005.

[37] S. Valero, E. Argente, V. Botti, JM Serra, P. Serna, M. Moliner, and A. Corma. Doe framework for catalyst development based on soft computing techniques. *Computers & Chemical Engineering*, 33(1):225–238, 2009.

[38] S. Vempala. Geometric random walks: A survey. *MSRI volume on Combinatorial and Computational Geometry*, 2005.

[39] Z. Zhou, Y.S. Ong, P.B. Nair, A.J. Keane, and K.Y. Lum. Combining global and local surrogate models to accelerate evolutionary optimization. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 37(1):66–76, 2007.

Appendix 1

A CD-ROM is attached to the thesis, containing:

- Commented source code of the prototype implementation (in the `source` directory and its subdirectories).
- Scripts demonstrating the use of the prototype implementation (in the `source/demonstration` directory).
- Data sets (in the `source/data` directory) and benchmark functions (in the `source/functions` directory) used to benchmark the proposed methods.
- This thesis in PDF format (in the `thesis` directory).